(1)

RL-TR-94-35
In-House Report
March 1994

1992 DISE SUMMARY REPORT

Vaughn T. Combs, Dr. Gary L. Craig, Francis A. DiLego, Jr., Jerry L. Dussault, David J. Gray, Patrick M. Hurley, Scott M. Huse, Anthony M. Newton, Jon B. Valente, Robert Vaeth

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

Rome Laboratory
Air Force Materiel Command
Griffiss Air Force Base, New York

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-94-35 has been reviewed and is approved for publication.

APPROVED: [illegible]

ANTHONY F. SNYDER, Chief
C2 Systems Division
Command, Control, and Communications Directorate

FOR THE COMMANDER: [illegible]

JOHN A. GRANIERO
Chief Scientist
Command, Control, and Communications Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE March 1994	3. REPORT TYPE AND DATES COVERED In-House

4. TITLE AND SUBTITLE
1992 DISE SUMMARY REPORT

5. FUNDING NUMBERS
PE - 62702F
PR - 5581
TA - 28
WU - 17

6. AUTHOR(S) Vaugh T. Combs, Dr. Gary L. Craig, Francis A. Di-Lego, Jr., Jerry L. Dussault, David J. Gray, Patrick M. Hurley, Scott M. Huse, Anthony M. Newton, Jon B. Valente, Robert Vaeth

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Rome Laboratory (C3AB)
525 Brooks Road
Griffiss AFB NY 13441-4505

8. PERFORMING ORGANIZATION REPORT NUMBER
RL-TR-94-35

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Rome Laboratory (C3AB)
525 Brooks Road
Griffiss AFB NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
Rome Laboratory Project Engineer: Patrick M. Hurley/C3AB (315) 330-2925

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)
The 1992 DISE Summary Report is a progress report on the activity of the Rome Laboratory In-House work in Distributed Processing Systems. In addition to the progress report, there are six consensed reports on the individual research, development, and application projects currently under way within the In-House group. The reports included are: Resource Management in Support of Application Management, Distributed-Parallel Effort, Distributed Image Compression in ISIS, Information Exchange using Cooperative Communicating Networks, The Survivable Distributed Computing Environment, and Weak Consistency and Recovery in a Command and Control Environment.

14. SUBJECT TERMS Distributed Operating System, Distributed System, Parallel, Consistency, Resource Management, ISIS, CRONUS, Distributed Computing Environment

15. NUMBER OF PAGES
52

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT U/L

Table of Contents

Foreword

This report is organized into three major sections. The Background section gives information on the history and rationalization for this manpower effort. The Progress Report section details the work performed under the 1992 fiscal year, beginning October 1991 and ending September 1992. The final section exists as a set of associated articles that describe some of the year's efforts. The articles are done in research paper format.

Background

The Command, Control, and Communications (C3) systems that the Air Force (AF) currently use are a collection of independent and interdependent systems. While sounding like a contradiction, the systems were usually procured for a particular function first, then integrated into the Command and Control (C2) structure later. Each system retains a measure of independence, but the information processed by the collection of systems is critical to our C2 measures and countermeasures. Our interest and thrust within this C2 environment are to provide the capability for all of the individual systems to be integrated into a distributed system (i.e. a system of systems). This entails having a set of heterogeneous hosts that communicate and manage one another in a timely fashion with fault tolerance, reliability, survivability, et cetera. This system needs to be capable of adjustment as new demands, capabilities, and technologies are integrated through software or hardware additions. Moreover, all of this must be done at a reasonable cost.

After realizing that the current level of research in Distributed Operating Systems (DOS) had reached a degree of maturity, the Computer Systems Branch of Rome Laboratory (RL) established an in-house capability called the Distributed Systems evaluation Environment (DISE) in fiscal year 1987. Within the DISE, we planned to continue the development of distributed systems technology areas and to provide a demonstration capability for our customers. The DISE seeks to provide a setting where: researchers can investigate and demonstrate the issues of concern for DOS technology, developers can design, create, experiment with, and verify distributed applications, and policy makers and technology managers can see the benefits of distributed technology.

Progress Report

At the start of our sixth year of operation, we had many ambitious plans. While not all of these plans materialized, we still managed to complete a major portion of them. This year we attempted 9 tasks, varying from applications development to research exploration. Of the 5 major projects, 2 were applications development, 2 were systems development, and 1 was research exploration. The remaining 4 minor projects were used to either complete tasks carried from the last fiscal year or explore interaction with other groups interested in distributed processing projects.

As discussed in our previous (1991) report, the Australian experiment is a multi-year effort to demonstrate the inter-operability of two allied communication networks on a cooperative basis to enforce national access/usage policies and to perform distributed command and control functions. Most of the effort during the fiscal year has been devoted to establishing a primary, demonstrable application and to planning the extension of that application into an environment partitioned into domains of control. By the end of the fiscal year, we had established the initial application and were completing the design of extensions for our new environment. More detail can be found in the paper entitled *Information Exchange using Cooperative Communicating Networks*.

Our distributed application instrumentation project, tasked to develop and design a tool for use within object-oriented distributed systems, was completed this fiscal year. A technical report on the project was published and should be available by contacting the address on the report documentation page. An overview of the system was published in our 1991 report. Unfortunately, we did not get to complete the user interface for the instrumentation tool.

A distributed-parallel systems task was created to investigate the affect of introducing parallel computers into a distributed computing environment (DCE) and to measure the feasibility of using a DCE to perform parallel algorithm tasks. Much of the initial work in this area has been dedicated to gaining an understanding of the synchronization relationships necessary to support parallel style algorithms. Synchronization and timing effects can be well understood when viewed from within a single computing device, but they become less understood when dealing with a set of independent computing devices connected within a networked structure. The *Distributed-Parallel Effort* paper describes our investigations within this area.

In order to increase the ability of object-oriented command and control (C2) systems to tolerate nodal and network failures, we established a research task to study consistency relaxation techniques. While we anticipated this task to develop, experiment, and demonstrate a new technique in loosening consistency requirements, a Congressional budget action terminated funding for the project. Before funding was withdrawn, we managed to finish an initial design for the system. This design is the basis for the paper entitled *Weak Consistency and Recovery in a Command and Control Environment.*

We wanted to demonstrate the effectiveness of distributed systems technology for helping C2 systems survive within hostile environments. So, we established the survivable C2 systems experiment as a multi-year vehicle for demonstrating a distributed information processing system that could adapt to sporadic and intermittent link and nodal failures while continuing its assigned C2 functions. After a year of debate, we finally agreed upon a general design to accomplish our intent. It is expected that a formal design and implementation will occur during the next fiscal year. The three papers *Resource Management in Support of Application Management, Distributed Image Compression in ISIS, and The Survivable Distributed Computing Environment* describe efforts under this task to better understand resource management issues, to learn about the ISIS DCE, and to establish a design for the system.

The Joint Directors of Laboratories (JDL) Tri-Service experiment is an ongoing application development project started in fiscal year 1989. Working with the Communications and Electronics Command and the Naval Research and Development laboratories, we have demonstrated the feasibility of integrating separate service applications for joint operations. During this fiscal year, we have been improving the survivability of our data management package along with updates to the X11 interfaces for the experiment.

For fiscal year 1993, we anticipate continuing work in our JDL, survivable experiment, and Australian experiment technology areas.

Resource Management in Support of Application Management

Dr. Gary L. Craig
Electrical and Computer Engineering Department
Syracuse University
Syracuse, New York 13244-1240

Vaughn T. Combs
Computer Systems Branch (C3AB)
Rome Laboratory
Griffiss AFB, New York 13441-5700

1.0 Introduction

Large distributed applications require a level of coordination or *application management* [1] to achieve a high degree of survivability and adaptability. This coordination includes such items as:

*Server group availability policy enforcement (for group based replicated servers) via *service availability management*[2],
*Application performance management, e.g. task or server migration, and
*Adaptation policy control.

In this effort we exploit a system-wide resource management facility to aid in the development and implementation of efficient distributed application managers (AMs). Our current interest in resource management is in the context of very large distributed systems. Such systems are assumed to be comprised of multiple, hierarchical interconnection networks. At the highest level, a system can be viewed as consisting of many "campus" centers or sites. For generality, it is also assumed that such a distributed system may span multiple policy domains. By its very nature, the system is heterogeneous both in topology and resource type. It is assumed, however, that except in the presence of failures, all nodes are reachable by all other nodes (the network reachability graph is not partitioned). This type of environment requires the resource manager to deal effectively and efficiently with failures and partitions of both an intra- and inter-campus nature.

In a prototype application manager, to be embedded within a fault-tolerant application, the AMs goals are:

1. to maintain system reliability in the presence of failures (by restarting failed managers at operational nodes).
2. detect performance bottlenecks and reconfigure the application on the distributed system accordingly,
3. provide a degree of fault-tolerance to non-replicated components of the application via a checkpointing and restart approach, and
4. coordinate recovery from more global failures such as network partitioning.

Such an application manager needs to be informed (or needs to determine) when any of the following events occur: 1) resource failure, 2) subsequent resource recovery, and 3) resource performance transition. These events are normally detectable system state events logged and managed by a resource management subsystem.

This points to a generic interaction between an application manager and a resource management subsystem. Below we specify an AM architecture, the Application Level Expert System

(ALEX). In section 4, a prototype version of ALEX (to be used to evaluate this architecture) is discussed.

then uses its rule base to react to the event.

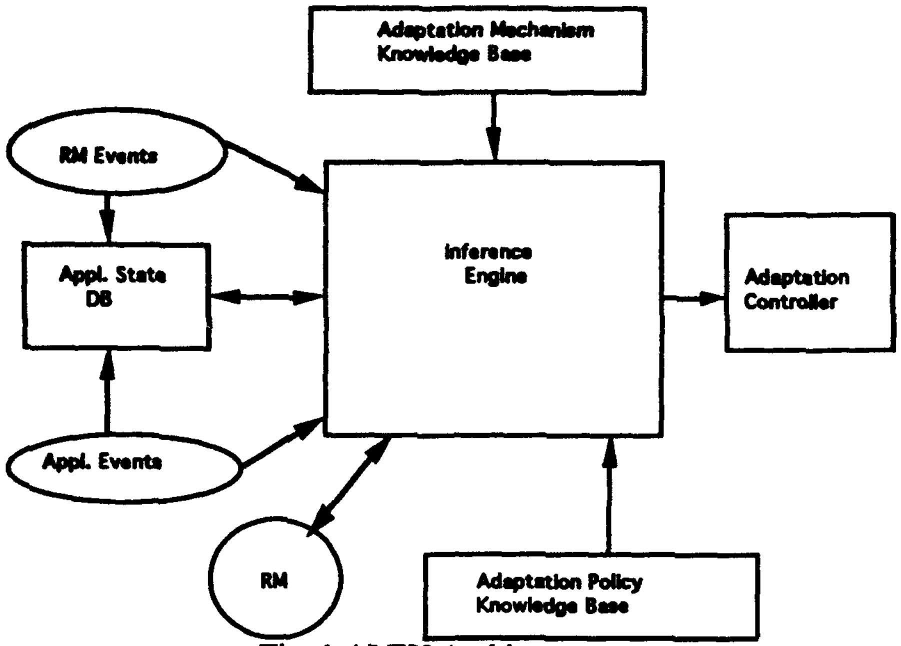


Fig. 1 ALEX Architecture

2.0 ALEX

The Application Level Expert System is an event driven management architecture which is designed to react to changes both in the underlying environment and within the application being managed. ALEX has been designed to be object-oriented. It encapsulates and maintains an object state which is representative of both the application and operational environment state. Only state information which is necessary to support the underlying control "machine" is maintained.

ALEX makes use of a resource management system, see Fig. 1. Predicates describing system state change "events", which involve resource state variables are defined and integrated within the resource manager. A corresponding operation interface (event handler) is also registered with the resource manager. When the predicate becomes true the RM invokes the specified operation, i.e., communicates the event occurrence to ALEX. ALEX then uses its rule base to react to the event.

The rule base is built by the application designer to effect the necessary control of the application depending on the event that has occurred. The application designer may also define certain application level predicates (events) that are of interest and specify the appropriate rule accordingly.

ALEX may interact with the application in one of two ways. The application designer may specify probes which signal application level events of interest. Alternatively, the application designer may export an interface to application components of interest, permitting ALEX to obtain specific state information. Note, the former organization is preferred. Should the state information cause a predicate to become true, ALEX, once again, consults its rule base in order to make the necessary user defined control decisions. Such an interaction is similar to the approach used in [1].

The above capabilities allow ALEX to react to changes in distributed system resources (processor failures, overloaded processors, communication failures, etc.) and dynamic changes in application parameters. For example, in a test application (JDL-DTE), ALEX may react to failed processors that may be running a mission critical replicated manager by automatically starting another one up on another node and bringing it to a state that is consistent with the other copies. ALEX may react to an event flagging the over utilization of a processor running key application components by requesting a resource profile involving all nodes that are currently running the application managers in question. Based on the returned data and its rule base, ALEX may then decide to migrate one or more application components to alternative node(s).

Given that ALEX represents a key component in an application, it must be a replicated service. As a distributed control system, a replicated ALEX must deal with the problem of distributed consensus. One possible strategy is to have each replica responsible for application control involving only a subset of the resources utilized by the distributed application. This hierarchical decomposition is similar to the RM architecture and minimizes the volatility of mutually consistent data.

The AM gets additional system state information from the RM (via query) to assist it in making control decisions (this interface is organized in this fashion to preserve encapsulation). The specific control actions are performed by the AM via both application specific mechanisms and mechanisms provided by the distributed computing environment.

A prototype of a scalable RM (supporting only the monitoring of individual node performance) has been implemented on top of the Cronus distributed computing environment [3] and is described in the next section. In addition, a prototype AM is being developed for a C3I application, the *Joint Defense Laboratory Tri-Service Distributed Technology Experiment*, (JDL-DTE)[4] and is described in section 4.

3.0 Prototype Resource Manager

The following section is a description of the underlying constituent components that make up a prototype resource management system. The resource management system monitors and collects certain fundamental system data and events that may be important for the continued efficient operation of survivable distributed application components. This is essentially the same information which is needed by the underlying system to facilitate system-wide resource management. In these two roles, an RM must have a highly efficient interface to be effective.

The RM must be scalable in order that it be successfully deployed in our distributed system. We have developed an RM Architecture which we believe is scalable within the framework of our distributed system model. It seeks to take advantage of *locality of service* to minimize the effects of *information latency*. It is the information management component of a RM which drives an AM, i.e., triggers events. An AM may then contract out for Administration services from either an RM, the distributed operating system, or some other applicable server(s). Thus, using the system resource information and contextual information known only within the application, the components of the application can be made to adapt (controlled by the AM) to underlying changes in the pool of available resources.

3.1 Architecture Components

The design of a prototype resource management system has been decomposed into three major components (Fig. 2.):

Resource Broker - The resource broker

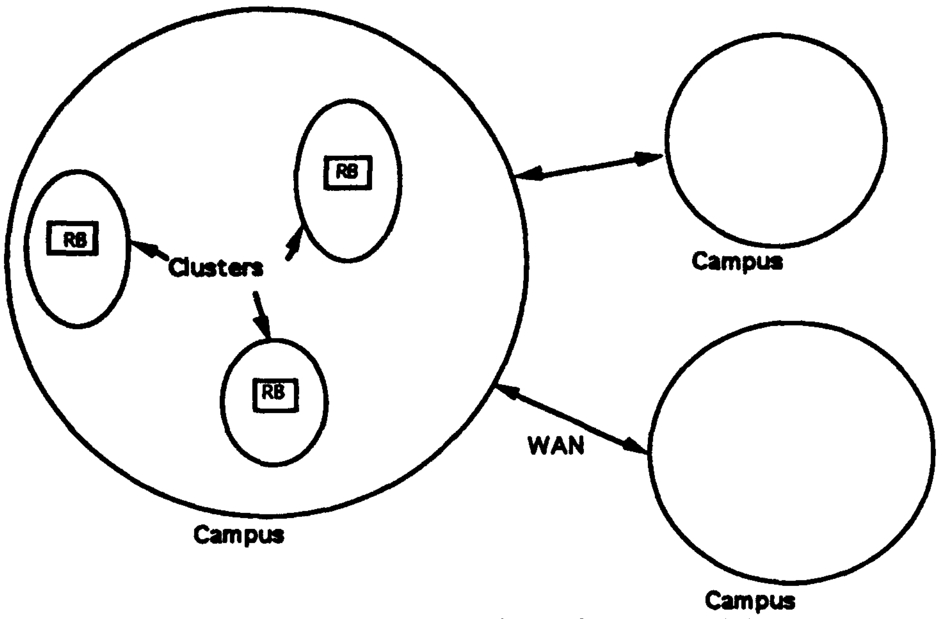


Fig. 2 Distributed System Model and Resource Manager.

(RB) maintains resource and service databases for a single domain (typically a LAN). Note that the collection of RBs in the system represent a distributed RM database. The contents of these global resource databases can be selectively accessed via query requests made upon a local RB. A request may be satisfied locally or may result in a request forwarding mechanism being invoked utilizing "peer brokers".

In our prototype, the components of the distributed application register themselves with an AM and hence with a RB. Applications make use of the information through the use of the AM (expert system) which makes application control decisions based on the data maintained and provided by the RBs. The AM at any time may query to determine any of the system status information being maintained by the collection of RBs. Note, this information will be used for control (reconfiguration) decisions by the AM.

Configuration Monitors - The resource configuration monitor (RCM) is hierarchical and is responsible for: 1) maintaining knowledge about the status and location of resource brokers in the system, 2) monitoring the health of system resources, and 3) maintaining (via the RB peer lists) a mapping of domains currently supporting a given service.

Resource Reporter - is assigned to each managed resource. The reporter is responsible for informing the resource broker for that resource when significant changes in the performance status of the resource occur.

3.2 RB interfaces

The main module of the resource manager is the Resource Broker. As such, considering the various interfaces of the RB is essential for understanding the RM. There are two major object types managed by the RB: Resources and Domains. A resource object encapsulates both the static and dynamic attributes of some physical resource, e.g., node. A domain (or service) object encapsulates attributes of a service. For each domain there is a list of (local) resources which support the service and a list of peer RBs which maintain mappings for additional resources.

Other than administrative interfaces, the following are important to our current project:
*GetTuplesforDomain(domainname, flag,*tuplelist);*

alterResourceCharacteristic(resourcename, status);

GetTuplesforDomain permits querying the current status of all resources which support a given domain. The flag permits restricting the search to *local only* resources. The *alterResourceCharacteristic* is the entry point that a resource reporter uses to update the RB resource database. The RCM also uses this interface to signal that a resource is suspected to be unavailable.

Because of limitations in Cronus, we cannot provide a generic predicate registration facility in RB[1]. Thus for prototyping, the RB has a single known entry point for a single AM (namely the JDL-DTE AM) to signal a resource:

1. becoming unavailable,
2. crossing above a high water mark (performance metric), and
3. falling back below a low water mark.

4.0 Prototype Application Manager

In Section 2 we described, in very general terms, an AM that would utilize the RM to make distributed applications more survivable and run more efficiently. The following discussion describes our prototype AM, being deployed within the JDL-DTE, in more detail.

The JDL-DTE is a distributed target tracking scenario which is comprised of eleven cooperating services (object managers). In order to provide a degree of fault-tolerance (survivability), several of the services are replicated. Cronus and replicated services automatically handle single manager failures transparently. However, it is necessary to provide an intelligent controller to determine when and where to start-up an additional "replacement" manager. The JDL-DTE AM is also responsible for improving application throughput by migrating managers away from "bottleneck nodes".

The prototype AM manages three interacting object types: ServiceLevel, Nodes, and Services. The ServiceLevel database maintains a set of prioritized application configuration objects, which specify application requirements e.g., the number of active copies of each application service. A node object is the AM's view of the system state, maintaining the resource's status (DOWN, LOWWATER, HIGHWATER), and its service status (Supported/Unsupported, Active/Inactive). A service object performs the mapping of a given service to supporting nodes and where it is currently active.

The interface for the RB to register events is:

modifyResourceStatus(nodename, status);

This interface indicates a change (discrete representation) in a resource's status. The AM can then infer what effect the resource status change has on the application. For example, the event resource A marked as DOWN implies the loss of all services previously active on that node. The AM must then determine (based on the ServiceLevel policies, and system status) what corrective action should be pursued.

Other than the policy specification and threshold values, all of the static data values are obtained through interaction with Cronus services (hostdata and servicedata databases, and each host database). Likewise, Cronus services are currently used to effect application

[1] Communication stubs must be linked into the RB manager's executable in the Cronus architecture.

configuration change (starting and stopping managers).

An integral part of this prototype development is likely to be the extension of the JDL-DTE's status display. The AM provides an excellent window for viewing the interaction of the application's status with the operating environment. For example, it is possible to see a node's performance deteriorate followed by the subsequent application reconfiguration to adapt to the change. Further, if desired, it may be possible to display the particular set of rules used to affect the application's reconfiguration.

References

[1] K. Marzullo, R. Cooper, M. Wood, and K. Birman. Tools for Distributed Application Management. *IEEE Computer*, 1991.

[2] F. Cristian, Understanding Fault-Tolerant Distributed Systems, *Communications of the ACM*, 1991.

[3] R. Schantz, R. Thomas, and G. Bono, The Architecture of the Cronus Distributed Operating System, *6th Intl. Conf. on Distributed Computing Systems*, pp. 250--259, May 1986.

[4] M. Gadbois and A. Newton, Tri-Service Distributed Technology Experiment, *Proceedings. IEEE/JDL Symposium. on Command and Control Research*, 1990.

Distributed-Parallel Effort

Jon B. Valente Robert Vaeth
Computer Systems Branch (C3AB)
Rome Laboratory
Griffiss AFB, NY 13441-5700

1. Overview

This effort is part of an in-house program that is looking at some of the issues that are associated with incorporating parallel computers into distributed systems and with doing parallel computing with a distributed system.

The program is made up of the following tasks:

Task A: Communications,
Task B: Program Definition,
Task C: Homogeneous Systems, and
Task D: Heterogeneous Systems.

This paper describes the results of work associated with Task C. The work performed is a joint effort between Rome Laboratory (RL) and the Northeast Parallel Architecture Center (NPAC) at Syracuse University. The two organizations form a two node network on which the experiments will be performed. NPAC's tasks involve working on the application and participating in the experiment. RL's tasks involve monitoring the experiment, acquiring the necessary hardware, performing the experiment, and performing the data analysis.

2. Breakdown of the Effort

The Task C work is grouped into two primary sections: the Distributed-Parallel Experiment and the Global Time Transformation. The work associated with the Distributed-Parallel Experiment follows these parameters: define and implement a set of experiments, define the metrics for the experiments, take measurements, develop a data storage and retrieval structure, and develop an analysis capability. Global Time Transformation addresses the problem of transforming the timing information from four different clock sources into the timing information that would have appeared from a single (global) clock source. This effort became necessary after a preliminary analysis into the experiments proposed by the Distributed-Parallel experiments.

3. Distributed-Parallel Experiments

The Distributed-Parallel Experiment is the main section of the effort. The work activities are broken up into the following subsections

1) Problem Definition,
2) Metrics and Measurements,
3) Storage and Retrieval System, and
4) Analysis.

The Problem Definition subsection defines a scope for the experiment, an implementation strategy, the hardware requirements, and the software requirements. The Metrics and Measurements subsection determines the metrics needed for the analysis subsection and the measurement techniques needed to capture them. The Storage and Retrieval System handles all the issues that deal with the data storage, backup and retrieval. The Analysis subsection is where all the data analysis is performed and the results of the experiment are generated.

3.1 Problem Definition

The problem definition section of the effort actually scopes out the experiment, develops an implementation strategy and defines the hardware and software requirements.

The implementation strategy focuses the effort, keeping it from pursuing secondary issues. The hardware and software requirements are derived from the experiment by looking at what is required and what is available. An adequate and significant amount of work is assigned to this section in order to ensure the success of the effort.

The majority of the problem definition work was accomplished in meetings between NPAC and Rome Laboratory.

3.1.1 Scope

This effort studies the effects of a two node network on the execution of a parallel application. The representative C^3I application uses networks located at Syracuse University and Rome Laboratory. The observations for the experiment are made to characterize the nature of application communication.

3.1.2 Implementation Strategy

The experiment executes in a two node network environment between RL and NPAC (See Figure 1). The two nodes connect through a T1 line, with network monitors at both ends to capture the messages. Multiple experiments are run to eliminate any anomalies.

The application is implemented in four different execution profiles: sequential only, parallel only, distributed sequential, and distributed parallel. NPAC is responsible for all work associated with the application.

RL is responsible for all the Local Area Network Protocol Analyzer (LANPA) programming, the data analysis, and the Global Time Transformations.

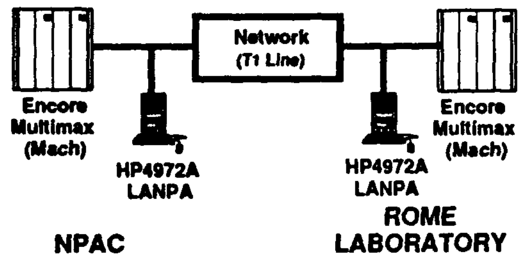


Figure 1

3.1.3 Hardware Requirements

The hardware used to run the experiment consists of two computers, a network, and a pair of network protocol analyzers.

The Encore Multimax was chosen as the default computer because of its availability at RL and NPAC. There is no additional requirement for computer system support since software, internal to the Multimax, exists to acquire and store the data.

The hardware needed to record the message passing between the computers on each network consists of the two LANPAs (see Figure 1). The LANPA selected for use is the HP4972A because it meets the requirement and is already in use at RL. The two networks that directly connect to the Multimax computers are, in turn, connected by organizational networks at Syracuse University and Rome Laboratory. Both Syracuse University and Rome Laboratory connect through subscription to T-1 class networking services provided by a local carrier.

3.1.4 Software Requirements

The software requirements are in seven categories:

1) the resident operating system,
2) the distributed operating system,
3) the application software,
4) the measurement software to be embedded into the application,
5) the control, measurement and support for the LANPA,
6) the software to upload the data from HP LIF format to the Multimax UNIX format and
7) the data analysis software.

Cronus, a distributed computing environment toolkit, is used to distribute the application. MACH is the operating system for the Multimax because the Cronus implementation for UMAX 4.3, the resident operating system, does not use the concurrent thread package. The Cronus implementation for MACH does not have this limitation.

The selected application is a concurrent multiple target tracking program from the California Institute of Technology at Pasadena (see Figure 2). It is better suited to the concurrency model of computing that is supported by Cronus than other candidates that were considered. NPAC installs the measurement software for the application.

RL generates the control, measurement and support software for the LANPAs, as well as the software to upload the HP LIF files from the LANPAs. The data analysis software is broken up into three areas, data retrieval, data transformation (i.e. global time transformation, GTT), and analysis.

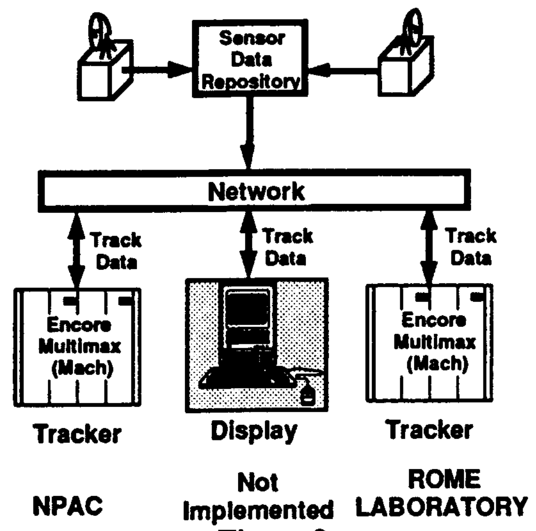


Figure 2

3.2 Metrics and Measurements

Three metrics measure the ethernet frames: network propagation delays, clock offsets, and arrival/departure times. Each LANPA monitors, time stamps, and records messages which originate and arrive at each Multimax. Clock offsets are calculated in two places: between the two Multimax computers, and between a Multimax computer and its local LANPA.

Software within each application program and LANPA collects the measurement data. In the application, software probes are present at those points where frames or messages send data to the network. These probes produce the timestamp information.

The LANPAs need both hardware modifications and new software to record the network activity of interest. Examples of this software include a new set of lists, filters, messages, and control programs. Furthermore, another program converts LIF data (the standard format for LANPAs) into ASCII text.

3.3 Storage and Retrieval System

The storage and retrieval system is the data link between the experiment and the analysis. The storage and retrieval system contains all the data associated with the activities of the experiment as they occurred in time. This allows us to reconstruct, if necessary, the sequence of events that make up the experiment.

The storage system consists of the media required to store primary and backup copies of data, and the translation software that converts LIF format data to ASCII text. The retrieval software accesses the ASCII text for use in analysis.

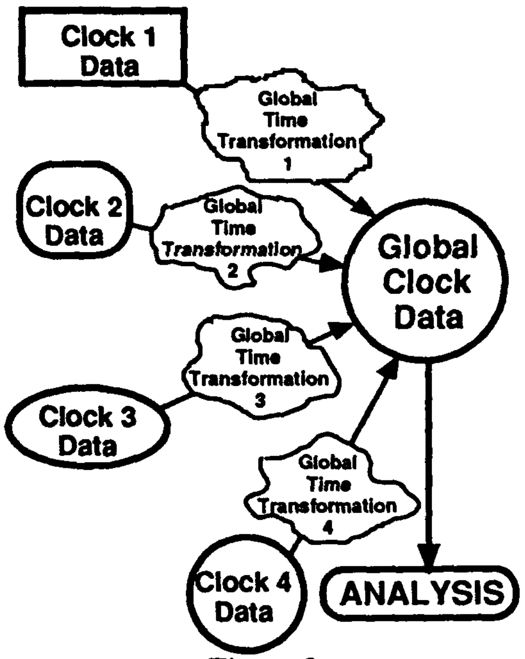
3.4 Analysis

The analysis work includes the calculations for speedup, execution time to communication time, and clock offsets which are used in the Global Time Transformations.

The analysis software gains access to the data by using the storage and retrieval system. The most complex problem in the analysis is the data merging and data association. There are four separate data sources for each experiment. Many problems may arise such as communication system breakdowns, lost data entries, and multiple data entries for the same message.

4. Global Time Transformations

The experiment does not use a distributed or global timing system. The two computers and two LANPAs generate time stamps based on the local clocks for each device. Without synchronization, it is difficult to derive useful information from the data. The Global Time transformations correct this problem by mapping each time stamp into the appropriate clock value of the global time clock (see Figure 3). The Global Time transformations involve several components: the derivation of appropriate analytical transforms, the calculation of drift coefficients, and the measurement of local clock offset values.


Clock 1 Data
Global Time Transformation 1
Clock 2 Data
Global Time Transformation 2
Global Clock Data
Global Time Transformation 3
Clock 3 Data
Global Time Transformation 4
Clock 4 Data
ANALYSIS
Figure 3

4.1 Transformation Definition

As a main objective of the transformation, the work involved in defining the transformation derives the appropriate analytical function and proves that the derived functions are valid.

To determine the appropriate analytical transformation, project staff plot the observations of the timed values generated by each of the four computer systems. Curve fitting programs determine the nature of the difference in clock time values.

4.2 Clock Drift Coefficients

The clock drift coefficients are numeric values calculated to adjust the measured time stamps recorded from the internal clocks of the 4 computer systems. Experiments determine if the coefficients are constant values. Project staff then develop a methodology to measure the clock drift by individually determining the drift rate for each clock in the experiment and then calculating the overall drift rate.

4.3 Clock Offset

It is very difficult to align the internal clock of each computer system at the start of an experiment, without a mechanism for global synchronization. To compensate, a clock offset is measured and used in the transformation process.

Since the LANPAs are not able to generate dynamic messages, project staff developed a technique for measuring the clock offset that used pr [illegible]fined messages. The LANPA techni[illegible]e uses two sockets to send and receive data. One socket receives clock based data at predefined time intervals while the other socket measures the propagation delay in the network. Calculation of the propagation delays encountered between the Multimax computer and the LANPA is also performed.

4.4 Transformation Verification

The algorithm used in the Global Time Transformation should be verified. This process involves two experiments: one to calculate the input data for the transformation, the other to provide comparison time data. The verification process, therefore, is defined by the following procedures:

1) define the Global Time Transformations for the 3 clocks,
2) get a set of drift data that is different than the data that was used to derive the Global Time Transformations,
3) plot both the Global Time Transformation predictions and the measured drift in order to compare, and
4) show that the error is insignificant for the amount of time that the experiment is running.

5. Progress

This report covers all work accomplished by 30 Sep 92. Most of the work required for the distributed-parallel experiments is complete. Both the problem definition and the metrics and measurements work are finished. The storage and retrieval systems are partially complete, with the remaining work to be finished in data retrieval. The data analysis is also partially complete.

5.1 Distributed-Parallel Experiments

All work under the Problem Definition phase is complete. The metrics and measurements work is complete.

The data analysis software has not been written, pending completion of the Global Time transformation work. Looking at the behavior of the application on differing computer configurations completes some of the initial work.

5.2 Global Time Transformations

Most of the work required to perform the Global Time transformation is complete. The transformation definition is done. The calculations for the clock drift coefficients and the clock offsets are close to completion. The verification process is also nearing completion.

5.2.1 Transformation Definition

Plotting the collected time stamp data, measuring drift, against the global reference clock yields a set of linear graphs. The linearity is verified through a variety of higher-order polynomial curve fits. The insignificant second and higher order terms within the polynomials verify that the graph is linear.

5.2.2 Clock Drift Coefficient

The clock drift coefficient work is almost complete. The experiments in drift rates lead to the derivation of a set of equations which filter the network effects from the clock drift coefficients. Using these equations provides a consistent set of results.

5.2.3 Clock Offset

In calculating the propagation delays between the Multimax computer and the LANPA, it is necessary to determine the response time of the Multimax and the LANPA. Work on this topic is underway.

5.2.4 Transformation Verification

The work involving transformation verification is partially done.

6.0 Conclusions

The data storage part of the system involved more work than we had planned due to the increased number of experiments and the volume of data encountered.

We developed an experimental capability along with the appropriate procedures for measuring and storing data.

The data retrieval system, the data analysis and part of the Global Time Transformation will be the only remaining work to be completed. When completed, two reports will be written: one for the Global Time transformation work, and the second for the Distributed-Parallel systems final report.

7.0 References

[Crow90] Crowcroft, J. and Onions, J. Network Time Protocol(NTP) over the OSI Remote Operations Service. RFC-1165, June 1990.

[Duda87] Duda, A., Harrus, G., Haddad, Y. and Bernard, G. Estimating global time in distributed systems, Proceeding of the 7th International Conference on Distributed Computing Systems, Sept. 21-25, 1987, pp. 299-306.

[Kope87a] Kopetz, H and Ochsenreiter, W. Clock synchronization in distributed real-time systems. IEEE Transaction on Computers Vol. C-36, No. 8, August 1987, pp. 933-940.

[Kope87b] Kopetz, H and Ochsenreiter, W. Interval measurements in distributed real time systems. Proceeding of the 7th International Conference on Distributed Computing Systems, September 21-25, 1987, pp. 292-298.

[Lamp78] Lamport, L. Time clocks and the ordering of events in a distributed system. Comm ACM, Vol. 21, No. 7, July 1978, pp. 558-565.

[Lamp85] Lamport, L and Melliar-Smith P.M. Synchronizing clocks in the presence of faults. Journal of the ACM, Vol. 32, No. 1, January 1985, pp. 52-78.

[Marz83] Marzullo, K. and Owicki, S. Maintaining the time in a distributed system. Proceedings of the 2nd ACM SIGACT-SIGOPS Symposium on the Principles of Distributed computing , August 1983, pp. 295-305.

[Milh45] Milham, W. Time & Timekeepers. New York, The MacMillian Company, 1945. pp 197-212.

[Mill89a] Mills, D. Internet time synchronization: the Network Time Protocol. RFC 1129, Network Working Group, October 1989.

[Mill89b] Mills, D. Network Time Protocol (Version 2) Specification and Implementation. RFC-1119, Network Working Group, September 1989.

[Mill92] Mills, D. Network Time Protocol (Version 3) Specification and Implementation and Analysis. RFC-1305, Network Working Group, March 1992.

[Ofek87] Ofek, Y and Faiman, M. Distributed global event synchronization International Conference on Distributed Computing Systems, September 21-25, 1987, pp. 307-314.

[Smar91a] Smari, W. Clock synchronization techniques: a survey. Final Report AFOSR Summer Facility Program, August 9, 1991.

[Smar91b] Smari, W. An overview of clock synchronization techniques. Technical Report AFOSR Summer Facility Program, August 1991.

[Smar87] Srikanth, T. and Toueg, S. Optimal clock synchronization. Journal of the ACM, Vol. 34, No. 3, July 1987, pp. 626-645.

Distributed Image Compression in ISIS

Scott M. Huse
Computer Systems Branch (C3AB)
Rome Laboratory
Griffiss AFB, New York 13441-5700

David J. Gray
Sterling Software, Inc., Intelligence & Military Div., KSC Operations
Beeches Technical Campus, Route 26N
Rome, New York 13440-2067

1.0 Introduction

1.1 Distributed Systems

In a distributed system, processing activities may be located in more than one computer, and the computers communicate over a network in order to perform joint tasks. At least three components of a system can be distributed. These include hardware, data, and control [Sloman].

A distributed system must consist of at least two computers, each with their own local memory and processors. The control strategy used to manage the resources of the system can be centralized, hierarchical, or completely autonomous at each site. The data can be distributed by replication or partitioning.

Whether it is necessary to distribute all three components of a system in order for it to be classified as a distributed system remains a debate. While there is no general agreement on the precise answer to this question, Enslow's model [Enslow] does require that all three aspects be distributed in order for a system to be classified as fully distributed. A simplified form of Enslow's cube model is provided in Figure 1. The degree of decentralization of a system is herein characterized by three dimensions which correspond to hardware, control, and data.

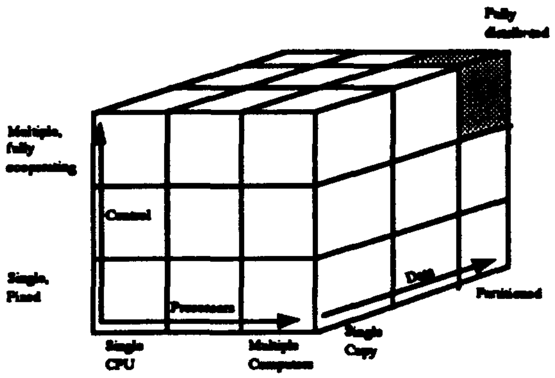


Figure 1. Enslow's Model of Distributed System Types

There are two key stimuli for the current interest in distributed systems: technological change and user needs [LeLann]. First, the growth in microelectronics, VLSI, etc., has altered the price-performance ratio to favor multiple low-performance processors rather than single high-performance processors. Also, interconnection and communication costs have fallen and numerous networks are now readily available and cost effective. Second, there are several potential advantages that distributed systems may offer over conventional computer systems which help to meet current user needs. These potential advantages include resource and information sharing, better reliability, flexibility and extendibility, modularity, better response and performance, and reduced incremental cost.

It should be noted that reliability, in particular, is a complex issue even in non-distributed computer systems. In distributed computer systems, issues such as correctness, fault-tolerance, and security become even more complex, and one must be concerned not only with the behavior of individual components, but also with their collective behavior in the overall context of the application. While one might expect the correctness of a distributed system to follow naturally from the correctness of its constituent components, this is not always the case [Birman, July 1991]. The ISIS system addresses this reliability concern through the notion of process groups and group programming tools which simplify the development of reliable distributed software.
The ISIS Distributed Programming Toolkit (version 3.0) is utilized in this research project. The ISIS toolkit was first released into the public domain in 1987 and has been distributed to more than 750 sites. ISIS is used in diverse settings including various banking applications, value-added telecommunication systems, wide-area seismic data collection and analysis, factory floor automation, education, and research [Birman, January 1991].

ISIS is a toolkit for distributed programming which provides a set of problem-oriented tools built around process groups and reliable group multicast. The ISIS toolkit provides for several different styles of process groups and also supports a collection of group communication protocols. In this research project, we utilize the client/server process group model and the cbcast (FIFO ordering) group communication protocol. That is, the client program interacts with the servers in a request/reply manner by multicasting to the server group.

ISIS implements a powerful model of distributed computation known as *virtual synchrony*. Virtual synchrony allows a programmer to write code while thinking of distributed events as occurring everywhere at the same time. ISIS enforces enough order so that the resulting code works correctly, while not sacrificing concurrency.

ISIS consists of language libraries and utility software which use Transmission Control Protocol/Internet Protocol (TCP/IP) and Unreliable Datagram Protocol (UDP) protocols [Coulouris, 1988] by default, and can be extended to use other protocols, e.g., MACH IPC. ISIS is available on a wide range of UNIX-compatible computers, and it supports application development in C, FORTRAN, and Lisp.

2.0 Research Overview

2.1 Introduction

In this research project, we have distributed the computationally intensive task of image compression using the Huffman coding algorithm. Coding required to reach our project goal includes implementation of both centralized and distributed versions of the same compression algorithm. Two factors necessitate the coding of a centralized implementation: first, to test our algorithm and learn how to manipulate image data; second, to identify the most computationally intensive steps in the image compression process.

From our centralized implementation we are able to decisively conclude that two steps of our algorithm would be ideal for distribution: counting of pixel frequencies and encoding of the image. Figure 2 provides timing results of our centralized implementation based upon a 1,037,012 byte 8 bit image file of 1152x900 pixels. From this large test image, we observed that these two procedures consumed approximately 94% of the total computation time. Timing results are obtained by running the centralized compression algorithm on a Sun SparcStation. Due to time constraints, no attempt was made to perform sophisticated performance measurements; rather, all results are based on simple 5-run averages.

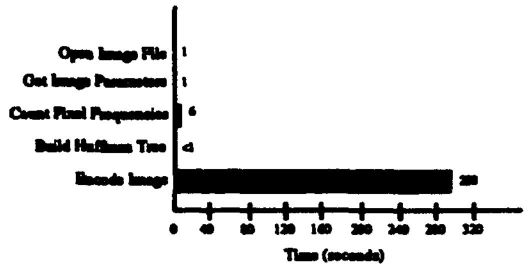


Figure 2. Centralized Compression Timing Results by Procedure

By distributing the counting of pixel frequencies and the image encoding stages, we are able to achieve a significant performance speedup. Details of the speedup obtained are provided in Section 3.0, Experimental Results.

In addition to the centralized and distributed implementations of our compression program, two supporting programs were also developed. These programs perform decompression of images compressed using the distributed implementation of our algorithm and also overcome a library conflict between ISIS and the SUN pixrect library. Decoding the image was done to verify the correctness of the processing performed.

For additional information on this research project, refer to Rome Laboratory Technical Report RL-TR-92-254. All source code developed for this project is included in this technical report.

2.2 Approach

Figure 3 provides a flow chart of the centralized algorithm implementation. Similarly, figures 4 and 5 provide flow chart illustrations of the distributed algorithm implementation.

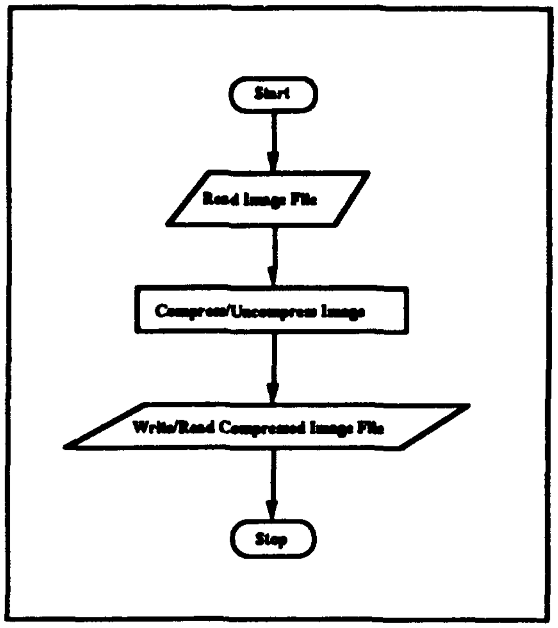


Figure 3. Flow Chart for the Centralized Implementation

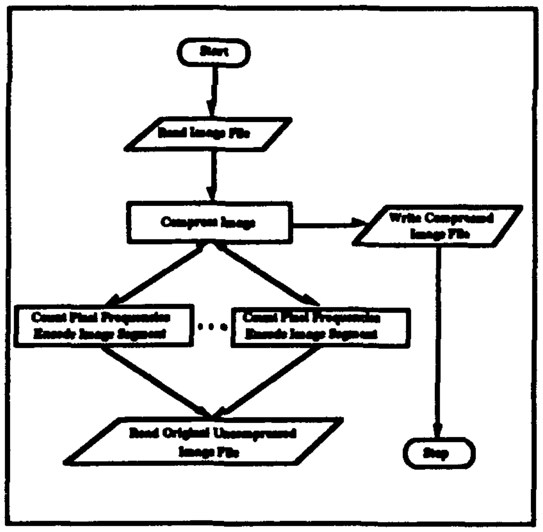

Start
Read Image File
Compress Image
Write Compressed Image File
Count Pixel Frequencies Encode Image Segment
. . .
Count Pixel Frequencies Encode Image Segment
Read Original Uncompressed Image File
Stop

Figure 4. Flow Chart for the Distributed Implementation

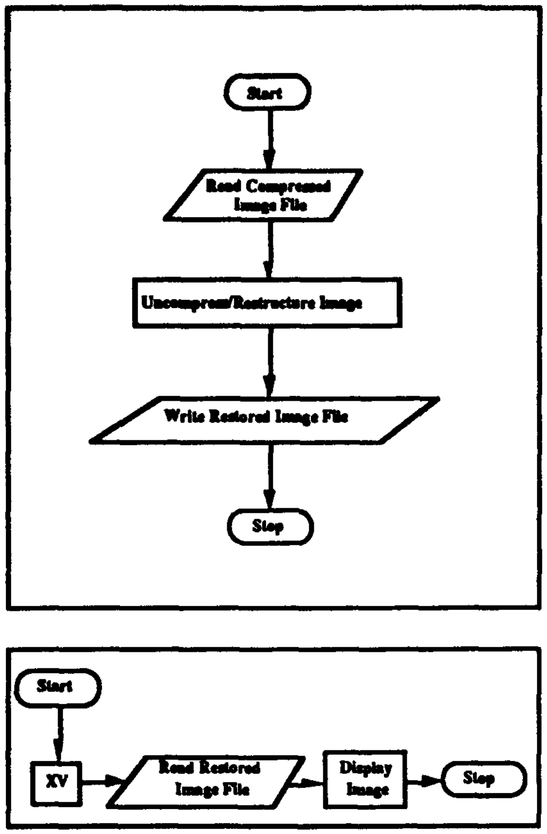

Start
Read Compressed Image File
Uncompress/Restructure Image
Write Restored Image File
Stop
Start
XV
Read Restored Image File
Display Image
Stop

Figure 5. Flow Chart for Image Reforming & Re-displaying

3.0 Experimental Results

In this section, we examine results obtained by experimentation of our distributed image compression algorithm. Our computing environment for all experimentation consists of a local cluster of SUN Sparc Workstations interconnected via an Ethernet LAN at Syracuse University.

Figures 6 through 13 provide timing results for eight test images. Each of these figures clearly shows the speedup obtained by varying the number of servers contributing to the image's compression. For each test image the number of servers was progressively increased until finally the number of servers caused the compression time to increase; that is, when *n* servers performed worse than *n*-1 servers.

Clearly, communication costs place a practical limit on the extent to which distributed image compression is beneficial. For our relatively small set of test images, the optimal number of servers varied from three to seven.

Each image's size and number of colors which it contains is provided with the image name in the figure captions. All of the plotted data points are based on 5-run averages.

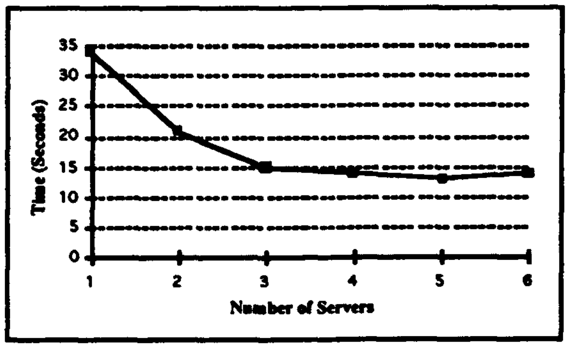


Figure 6. 637x436x8, 37 color image (Kuwait.ras)

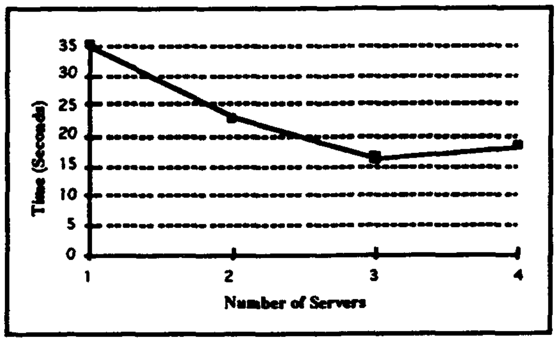


Figure 7. 727x436x8, 51 color image (Sukhoy27.ras)

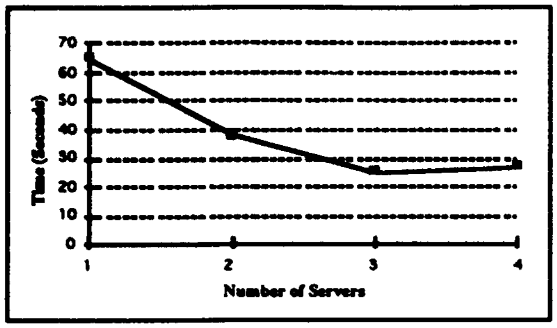


Figure 8. 618x435x8, 155 color image (Stealth.ras)

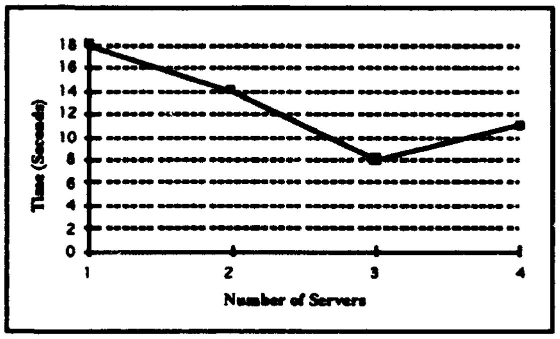


Figure 11 345x487x8, 32 color image (Einstein.ras)

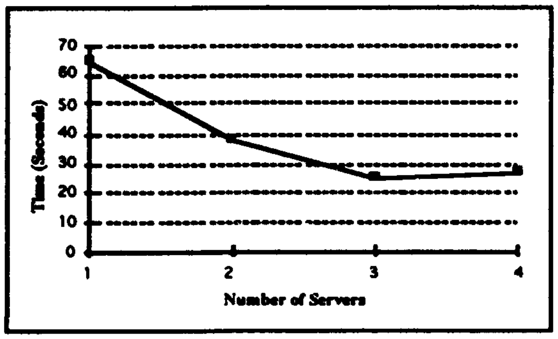


Figure 9 1152x900x8, 256 color image (Shuttle.ras)

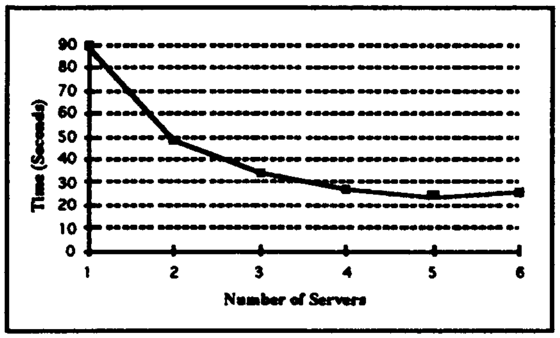


Figure 12. 512x438x8, 245 color image (F18s.ras)

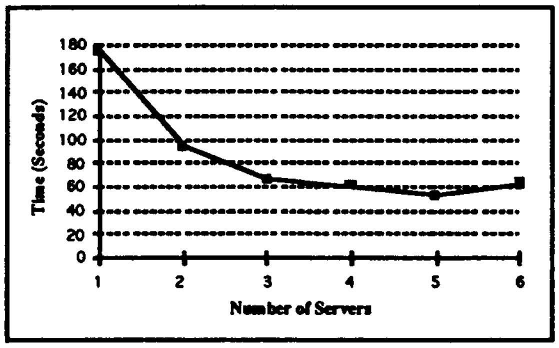


Figure 13. 792x538x8, 247 color image (F16s.ras)

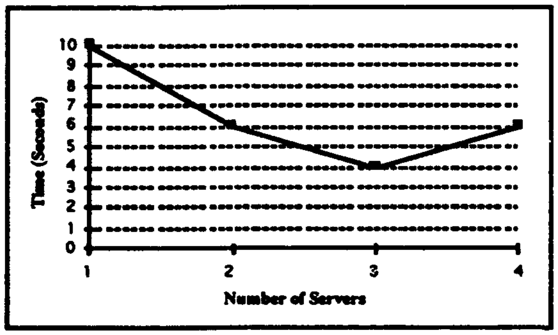


Figure 10 320x200x8, 32 color image (Billtcat.ras)

We may further demonstrate the speedup obtained by analyzing our timing results with the following simple formula:

$$\text{Speedup} = \frac{\text{Single Server Time}}{\text{Multi-Server Time}}$$

For example, we may calculate the speedups obtained for the various numbers of servers for the test image "Kuwait.ras" (see Figure 6) as follows.

1 Server: $\frac{27}{34} = .79$ **2 Servers:** $\frac{27}{21} = 1.29$

3 Servers: $\frac{27}{15} = 1.80$ **4 Servers:** $\frac{27}{14} = 1.93$

5 Servers: $\frac{27}{13} = 2.08$ **6 Servers:** $\frac{27}{14} = 1.93$

With such calculations for all test images, we may graph the relative speedup obtained through multiple servers compared with an optimal linear speedup. Figure 14 provides this graph.

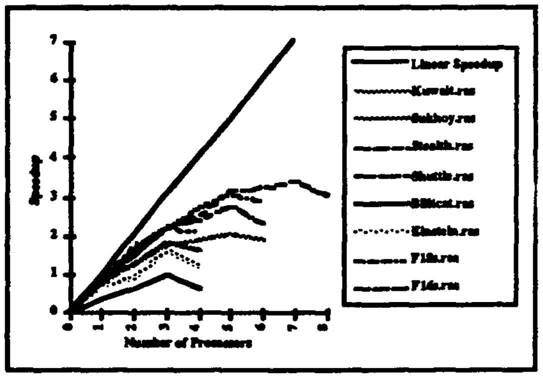


Figure 14. Multiple Servers vs. Optimal Linear Speedup

Given the known speedup supplied by varying the number of processors, we now consider the efficiency gained or lost through image complexity. Figures 15 and 16 provide timing results obtained from both the centralized and distributed implementations, respectively. Image statistics are also provided in these figures. In Figure 16, the total time given represents the processing time at what was found to be the optimal number of servers.

Image Data			Timing Results (seconds)			
Image Name	Dimensions	Distinct Colors	Pixel Frequency Counting	Image Encoding	All Other	TOTAL
Kuwait.ras	637x436x8	37	1	24	2	27
Sukhoy27.ras	727x436x8	51	2	27	1	30
Stealth.ras	619x435x8	135	2	33	1	36
Shuttle.ras	1152x900x8	256	6	296	2	304
Billcat.ras	320x200x8	11	1	3	0	4
Einstein.ras	343x487x32	32	2	11	0	13
F18a.ras	798x530x8	247	1	71	2	74
F16a.ras	512x438x8	246	2	143	3	148

Figure 15. Timing Results for the Centralized Implementation

Figure 17 provides processor load efficiency data. Efficiency is defined to be the average utilization of the *n* processors allocated to execute the distributed program. The efficiency of a single processor system, when ignoring input/output (I/O), can be equal to 1. In general, the relationship between speedup S(n) and efficiency E(n) can therefore be defined as:

$$E(n) = S(n)/n$$

,where n is the number of processors.

If efficiency remains at 1 as processors are added, S(n) equals n and linear speedup is achieved. This is an ideal situation in which incremental improvements in speedup can be made without loss of processing efficiency. In general, linear speedup is not achievable because of the contention for shared resources, process communication time, and so forth [Michaud].

Figure 17 provides the processor load efficiency data for our experiment based upon what was found to be the optimal number of processors. For example:

$$E_{\text{Stealth.ras}}(n) = E_{\text{Stealth.ras}}(3)$$

$$= \frac{2.24}{3}$$

$$= .75$$

Image Data			Timing Results (seconds)	
Image Name	Dimensions	Distinct Colors	Number of Servers	TOTAL
Kuwait.ras	637x436x8	37	5	13
Sukhoy27.ras	727x436x8	51	3	16
Stealth.ras	618x435x8	155	3	25
Shuttle.ras	1152x900x8	256	7	88
Billcat.ras	320x200x8	11	3	4
Einstein.ras	345x487x32	32	3	8
F18s.ras	792x538x8	247	5	24
F16s.ras	512x438x8	246	5	53

Figure 16. Timing Results for the Distributed Implementation

Image Name	Processing Time (seconds)		Efficiency
	Centralized	Distributed	
Kuwait.ras	27	13 (5 Servers)	.42
Sukhoy27.ras	30	16 (3 Servers)	.63
Stealth.ras	56	25 (3 Servers)	.75
Shuttle.ras	301	88 (7 Servers)	.49
Billcat.ras	4	4 (3 Servers)	.33
Einstein.ras	13	8 (3 Servers)	.54
F18s.ras	74	24 (5 Servers)	.62
F16s.ras	148	53 (5 Servers)	.56

Figure 17. Load Efficiency Data

Compressions percentages are summarized in Figure 23. Figure 24 graphs the relationship between the number of distinct pixel colors and the degree of compression.

Image Name	Unique Colors	Image Size (bytes)		% Compressed
		Uncompressed	Compressed	
Kuwait.ras	37	278,968	128,325	54%
Sukhoy27.ras	51	318,208	136,829	57%
Stealth.ras	155	269,630	207,615	23%
Shuttle.ras	256	1,037,600	964,968	7%
Billcat.ras	11	64,065	18,578	71%
Einstein.ras	32	168,630	94,433	44%
F18s.ras	247	221,984	208,665	6%
F16s.ras	246	426,896	342,882	19%

Figure 18 Image Compression Results

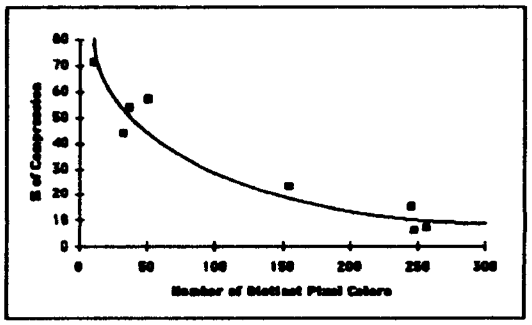


Figure 19. Relationship Between Distinct Pixel Colors & % Compression

Clearly, as the number of distinct pixel colors increases, compression ability generally decreases accordingly.

4.0 Final Remarks

In this research project, Sun raster images are successfully compressed using a distributed implementation of the Huffman encoding algorithm in ISIS 3.0. Compression percentages range from 6% to 71%. This range reflects the varying degrees of image complexity (primarily image size and number of distinct pixel colors) for our test images.

Most significantly, we find that by distributing two key modules of the algorithm (count pixel frequencies and encode image), speedup increases of up to 3.42x can be achieved while maintaining processor load efficiencies as great as 75 percent. We also note that communication costs place a practical limit on the extent to which distributed image compression is beneficial. This limit is found to vary significantly depending on the complexity of the particular image. For our relatively small set of test images, the optimal number of servers varied from 3 to 7. Clearly, distributed image compression algorithms can offer superior performance over centralized implementations, particularly for large and complex images.

References

Birman, Kenneth P., The Process Group Approach to Reliable Distributed Computing, *Technical Report TR-91-1216*, Cornell University Computer Science Department, pp. 1-35, July 1991.

Birman, Kenneth P., Robert Cooper, and Barry Gleeson, Programming with Process Groups: Group and Multicast Semantics, *Technical Report TR-91-1185*, Cornell University Computer Science Department, pp. 1-23, January 1991.

Coulouris, George F., and Jean Dollimore, Distributed Systems: Concepts and Design, Addison-Wesley Publishing Company, p. 101, 1988.

Enslow, Philip H., Jr., What is a "Distributed" Data Processing System?, *Computer*, pp. 13-21, January 1978.

LeLann, G., Motivation, Objectives, and Characterization of Distributed Systems, in Lampson, 1981.

Michaud, M. C., and J. B. Goethert, An Evaluation of Processing Efficiency on Multiprocessor Architectures: Volume 2, *RADC-TR-90-436*, pg. 4, December 1990.

Sloman, Morris, and Jeff Kramer, Distributed Systems and Computer Networks, Prentice-Hall International, pp. 1-7, 1987.

The Isis Distributed Toolkit (version 3.0) - Users Guide and Reference Manual, Isis Distributed Systems, 1992.

Information Exchange using Cooperative Communicating Networks

Francis A. DiLego, Jr. Jerry L. Dussault Anthony M. Newton
Computer Systems Branch (C3AB)
Rome Laboratory
Griffiss AFB, NY 13441-5700

1. Overview

The application design discussed within this paper is a subset of broader work produced by the Australian Experiment, an unclassified activity initiated under the auspices of the 1988 Memorandum of Understanding (MOU 88/102) "Cooperative Communicating Networks". The United States partially funds this activity under the Nunn Amendment. Rome Laboratory (RL), in the United States, and the Electronics Research Laboratory (ERL), in Australia, are the two participating agencies.

The Australian Experiment seeks to: (1) establish new networking technology based on policy-based routing [2], (2) implement a distributed application across multiple networks in the US and Australia, and, (3) experiment with the issues involved in sharing information and resources in a command, control, and communications environment.

2. The Environment

One of the problems with current, joint allied operations is the (clumsy) way in which data is exchanged. Differing data formats and classifications make direct information transfer difficult at best. The environment that we are creating would allow for the transparent transfer of data among multiple applications within a distributed environment.

2.1 Cronus Distributed Environment

BBN Systems and Technologies Corporation develops *Cronus* [1] under sponsorship of Rome Laboratory and other government agencies. Cronus is an environment for the development and operation of distributed applications. It provides the applications programmer with a set of tools that teaches a coherent and integrated systems approach to the development of computer applications which will be spread among several computing resources. Cronus runs on a variety of heterogeneous hardware bases and operating systems. On each computer system, Cronus operates as a set of user-level processes within the native operating system. By executing at this level, application developers can utilize both the support facilities of their local computing environment and the system integration capabilities of Cronus.

By using an object-oriented approach, application components or modules within the Cronus environment are highly transportable. Application components are called *managers*. Each manager is a separate self-contained process, which is responsible for maintaining and manipulating the data (a.k.a. *object*) entrusted to the manager. Each object has an address (a.k.a. *a unique identifier*) through which it can be contacted. Managers accept requests by other managers or clients to manipulate objects. The requests trigger *operations*, a pre-defined sequence of instructions which perform the object manipulation. Managers can be relocated or replicated at different computers by recompiling the source code. Once compiled and executed, a manager can be seen and reached by every other networked computer system that is running the Cronus software environment. Managers are also the only means of providing access to the object. Nothing, other than the manager responsible for an object, can access the object directly. Any client or manager that wishes to access an object must use the operation interface defined and supported by the object's manager.

2.2 Policy - Based Gateways

The policy-based gateway [3] functions as an Internet Protocol (IP) Router for an International Military Internetwork (IMI). Expressed differently, the IMI is formed by interconnecting national networks using policy-based gateways. These gateways are physically formed by two processors (one connected to each national network), with a data link between them. Like any typical Internet gateway, their primary function is to route IP packets between networks using a strategy that minimizes the number of "network hops" from source to destination. The policy-based gateways are unique in that they implement routing algorithms that incorporate a dynamic "routing policy" into their routing decisions. Thus, factors such as security, priority, classification, sensitivity, etc., may be combined to form a set of constraints (i.e., a policy) defining the conditions for transmittal and delivery of information.

2.3 DCE-Net Management Interaction

Existing computer systems use the traditional layered approach to distinguish between application, operating system, and network software. This layered approach has been limited in scope to the transmission of data from one layer to the next with little, if any, control and status information exchanged. With the development of intelligent network management and distributed computing environments, information collected by both network management and the computing environment could be shared for mutual benefit. As an example, consider the effect of mutual knowledge regarding network bandwidth. The distributed computing environment could perform resource (data) management more efficiently if it has information about the current sustainable bandwidth between sites. This would allow the resource manager to migrate data around to satisfy the timeliness requirements that applications need. Similarly, network (link) management could be more efficient if it has information about the current, and possible future, bandwidth demands between sites. This would allow the network manager to reconfigure links between sites to cover the current demand and future growth.

The resultant merger between network management and distributed computing tools creates a rich environment. In addition to the standard topology, bandwidth, and congestion control issues, the network is controlled by policy routing gateways. This allows the network manager to set broad policies (control information) which control routing based on parameters such as source-destination pairing, packet content, networks traversed, and security levels (these policies will be discussed with further detail in later portions of the paper). The distributed computing environment, having some knowledge of the policies in effect, can take action designed to affect performance, reliability, and the survivability of the entire computing environment at both system and application layers. Currently, the distributed computing environment is scheduled to inject content-based information, such as data types and security levels, into the packet before it is transmitted through the network environment. Network policies are enforced through software mechanisms contained within the policy routing gateways. These mechanisms operate on a per-route basis to determine the suitable routing path (hops) through the networks and domain structures. Some sample mechanisms are contained in the following paragraph.

2.4 Policy Mechanisms:

The following list of candidate policy mechanisms was developed and refined during FY92 through a series of discussions among Rome Laboratory and SRI personnel[1] .

1. *Source-Destination.* Gateways will allow/disallow traffic through a network (administrative domain) based on the source-destination address pair.

2. *Entry/Exit Domains.* Gateways will deny traffic based upon the next (entry) administrative domain, or the previous (exit) administrative domain in the source-destination route.

3. *Domain Traversed.* Gateways will deny all traffic that has already traversed a specific administrative domain.

4. *Domain to be Traversed.* Gateways will deny service that is destined to traverse a specific administrative domain.

5. *Source-Destination User Identification.* This is an extension to the source-destination policy mechanisms discussed above; however, service will be granted or denied based upon a specific source-destination process user identification.

6. *External Conditions.* Routing policies may be based upon some external input such as a threat condition (THREATCON) level.

7. *Utilization Schedule.* Routing policies may be based on use of a domain only during specific (e.g., off-peak) hours.

8. *Precedence (priority).* Routing policies may be based on some precedence level set within the message header.

9. *Traffic Type.* Similar to the precedence policy mechanism above, the gateways will make routing/network access decisions based upon a traffic type designation within the message header.

10. *Security.* Similar to the two policies above, routing and network access will be based upon the security classification of the message (indicated in the message header).

11. *Congestion.* Routing and access decisions would be based upon the current loading within the administrative domains. Implementation of these mechanisms assumes that a method is developed to detect a congested condition, and to inform the gateways of the network status.

12. *Resource Usage Limits.* Routing and access decisions will be based upon a user quota system (e.g., a limited number of packets per month). Once a user has reached their allocated quota, no additional service would be granted.

The purpose for developing this set of mechanisms was to give network managers a variety of options and considerable flexibility in controlling the access to, and utilization of their networks. While each mechanism could be used individually to implement a network control policy, we anticipate that the power of these mechanisms will come from the myriad of combinations that can be composed to implement very sophisticated and dynamic network management and control policies.

Of the policy mechanisms proposed above, a limited set was selected for implementation in the first release of the Cooperative Gateway software. This was primarily due to budget constraints. Based on their relative usefulness and cost to implement, only four mechanisms will be realized in the initial version of the Cooperative Gateway: source-destination, external conditions, traffic type, and congestion. These

[1] SRI is under contract to Rome Laboratory for the development of the Cooperative Gateway.

mechanisms are scheduled to be delivered during the first quarter of FY93.

2.5 Information Domains

Within the networking environment, traffic flows around and within community boundaries. Domains operate by placing structure around these natural boundaries so that interdomain and intradomain traffic can be efficiently managed and controlled. The same concept of domains can similarly be used within the command and control environment. As part of this effort, the command and control application used will experiment with the concept of information domains, and the effects or benefits of information domains when used to support international force (troop) deployments.

While this work is not scheduled to occur until the 1993 fiscal year, its impact can be demonstrated by an example. Using the experiment's scenario of a cooperative effort between the Australian and United States governments, tracking information is being kept by both forces in a distributed environment where national computing resources are integrated to support the mission. Clearly, there exist two domains of information, each aligned along national boundaries. It is the intent of the experiment to utilize the resources of the distributed environment, communication, and data, to show that information flow between the domains can be controlled and managed in real-time to near real-time.

3. Demonstration Application

The application chosen for this set of experiments will simulate a distributed regional surveillance system. To represent this environment, each country will have a distributed set of software modules that will simulate a particular surveillance platform (e.g., an Over-The-Horizon radar or AWACS). The configuration flexibility afforded by Cronus will be used to perform experiments in resource management, fault-tolerance and synchronization.

Although this Demonstration Application is not intended to be a detailed model of any specific C2 system, the components described are representative of the major functional components found in many operational systems. Likewise and most importantly, so too is the information exchange that must take place between the components in order for the system to function correctly.

3.1 Application Scenario

The scenario depicts a hostile incursion into the island chain off Australia's northern coast. This would result in a need for increased surveillance of the region. The increased surveillance would be provided by numerous surveillance platforms including Australia's OTH radar, a US AWACS and a US Intelligence source (e.g., satellite). The transparent sharing of surveillance information between US and Australian forces will be demonstrated using the distributed system and Cooperative Communications Network. As the system's performance degrades due to network overloading or component failures, the system will be reconfigured to maintain adequate performance and insure continued access to surveillance data. The need for more dynamic interaction between system operators and network managers will be investigated.

The scenario will demonstrate: (1) distributed processing, (2) information access control, (3) transparent access to distributed data, and (4) information exchange using policy-based networks (i.e., routing enforced by policy-based gateways) which are subject to dynamic change. Both the American (US) and Australian force roles can be reversed.

3.2 Networking Topology

Each site will have an internal set of networks interconnected by policy-based gateways to simulate an independent national internet environment. The two internal national environments will be connected with a set of policy-based gateways to allow communication between the sites. The Cronus distributed computing environment will be installed on machines in both internets and will be used to create an application across both sites. Traffic injectors will be used to simulate varying system and network loads during the experiments.

3.3 Application Components

The application software designed for this experiment consists of six major components:

(1) Target Simulator Manager (TSM)
(2) Synchronization Manager (SM)
(3) Sensor Platform Manager (SPM)
(4) Filter/Correlator Manager (FCM)
(5) Mission Data Manager(MDM)
(6) User Interface Client (UIC)

It is important to note that neither the TSM nor SM are strictly part of the C2 Application, but are required to drive the demonstration. These two components provide the SPMs with the dynamic scenario activity (e.g., location of friendly and enemy units) scripted for each experiment (see Figure 1).

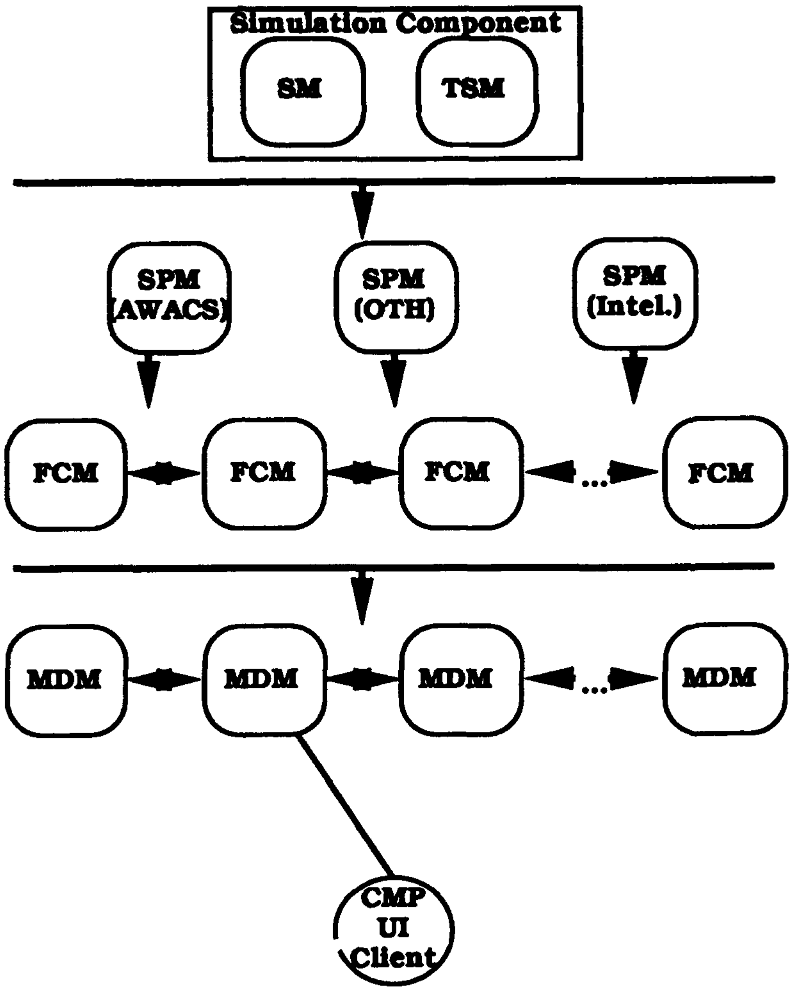


Figure 1

The SPMs obtain the scenario time and respective situation data from the TSM and SM and report on the target tracks they should be able to detect based upon their type. The software structure for a sensor platform will be configured at the start of the experiment to best represent a given surveillance platform (e.g., OTH Radar, AWACS). Multiple, independent sensory modules will report to a set of cooperating FCMs.

The FCMs will formulate and identify a target as being real or noise and calculate its bearing. Then the FCM will form a consensus with all other reachable FCMs to reconcile the most up-to-date sensor hits for any overlapping regions of coverage. When the most recent target information is established, the information is stored in all of the FCM's local databases. The information is also passed on to a Data Management Service (DMS) wherein the track history of the targets acquired by the various sensor platforms is maintained in a database repository.

Each database repository will be constructed with facilities for exchanging information between them. These database repositories will be accessed and manipulated by a Mission Data Manager (MDM). For survivability, the database repositories should also be replicated. The MDM's will exchange information to ensure replicated up-to-date target information is being stored. Separately, the policy-based gateways will arbitrate to see if the information can actually pass between sites given the current network topology and policy agreement. If the information cannot pass through the gateways, the MDM will de-replicate those objects within the databases that cannot be exchanged.

The user interface (UI) in this application acquires its information from the DMS. For this application we are using a version of the Common Mapping Program (CMP) [4] demonstration interface, which has been altered to act as a client to the Cronus distributed computing environment. The CMP is a development effort being sponsored by Rome Laboratory. This effort has several components of which three are being used here. They consist of DMA data (processed into CMPS Common Mapping Standard format), the Common Mapping Toolkit (CMTK), which is a library of functions to manipulate CMS data, and a demonstration user interface. CMP's main goal is to develop a standard format and tools for the many different forms of cartographic and imaging data. The incorporation of CMP into this experiment is based on the possible designation of CMP as an Air Force or DOD standard.

The UI demonstration software and embedded CMTK calls are implemented through MIT's X11/R4 and Motif 1.3. During FY92, the UI was constructed using a Sparc RISC-based Solbourne series 5 multi-processor. The existing UI had to be tailored to the needs of the experiment. This was done by removing unnecessary functionality, tailoring display of information and adding needed functionality. The added functionality that was most important was making the new UI a Cronus client and animating the target information. The Cronus object oriented programming paradigm made the task of modifying existing, single-system, stand-alone code easier. The addition of animated targeting information was also made easier because of the ease of using the CMTK.

4. Future Work

The most current goal is to complete the C^2 application's components (SPM, SM, etc.). The next obvious step will be the integration and testing of these components with the cooperative gateways and policy-based information exchange concepts, allowing us to analyze the interaction between the DCE and Automated Network Management (ANM) tool [5]. Then, through a series of experiments and demonstrations using a representative distributed C2 Application (discussed above), we will begin to assess the performance, utility and practicality of policy-based routing and specific policy mechanisms.

5. References

[1] Berets, James C., Mucci, Ronald A., and Schantz, Richard E., "Cronus: A Testbed For Developing Distributed Systems", IEEE Military Communications Conference, IEEE Communications Society, October 20 - 23, 1985.

[2] Bowns, H., Steenstrup, M.; "Inter-Domain Policy Routing Configuration and Usage."; Internet Draft; Internet Engineering Task Force; July 1991.

[3] Lee, D.S., Merchant, S., Lee, D.Y., Denny, B.; "Cooperative Gateway I: Software Design Document"; ITAD-2568-TR-92-108R; SRI International; Menlo Park, CA; October 1992.

[4] "Common Mapping Program: Users Manual"; Version 1.4.1; Sterling IMD, Inc; Rome, NY; Sept 1992

[5] Network Services Dept; "ANM: Advanced Network Management: Users Guide";BBN Doc #7920; BBN Systems Technology Center; Cambridge, MA; 1993

The Survivable Distributed Computing Environment

Patrick M. Hurley *Scott M. Huse*
Computer Systems Branch (C3AB)
Rome Laboratory
Griffiss AFB, New York 13441-5700

1.0 Introduction

Distributed computer systems support several key attributes that are essential for the development of command and control (C2) applications. The Distributed Systems Environment (DISE) is currently able to demonstrate and integrate many of these attributes. These include heterogeneity, replication, fault detection/recovery, and limited adaptive resource management.

In order for the development of C2 applications to become more survivable, more dispersed, and better able to quickly adapt to new threats, we are seeking to provide and demonstrate a Survivable, Distributed Computing Environment (SDCE). One of the primary attributes for the SDCE is *fault tolerance*. Fault tolerance may be defined as the ability of a system or component to perform its function, despite the presence of hardware or software faults. Fault tolerant mechanisms which detect and/or recover from hardware and software faults are essential for survivable systems.

In essence, the SDCE will be a base on which survivable distributed applications can be built. This base will be flexible enough to incorporate advances in technology. It will also be tailorable to the needs of specific C2 applications, and well-structured for ease of maintenance. Hence, this base will be capable of evolving with the needs of C2 systems and their supporting technologies.

2.0 SDCE Requirements

The Survivable Distributed Computing Environment must provide the underlying mechanisms to support the development of highly reliable distributed C2 applications. In order to accomplish this goal, a list was generated describing the requirements that should characterize the SDCE. However, due to time and personnel constraints, the SDCE in-house group will not be able implement all of these requirements. At a minimum, however, the SDCE must provide the underlying mechanisms that are capable of supporting all the specified requirements. Included in the list of requirements are distribution, fault tolerance, real-time, adaptive resource management, support for multi-domained applications, multi-clustered networks (interconnection of LANs and WANs), and heterogeneity (to include languages, operating systems and machine architectures).

The SDCE should support distribution because it provides many desirable attributes. These attributes include high availability of resources (to include data), improved reliability and increased performance (concurrent processing).

The SDCE should also support underlying fault tolerant mechanisms in hardware, software, and communication. Numerous types of faults in hardware, software, and communication could be considered. However, due to the complexity of dealing with some faults, the requirements for the SDCE will be limited to the following candidate list.

Two classes of hardware faults should be supported by the SDCE - (1) the complete loss of a host/service; and, (2) degradation in the performance of a host/service due to overload conditions. In terms of software faults, the SDCE should address - (1) algorithmic faults (perhaps through support of n-version programming); (2) software component faults (service loss); and, (3) timing faults. Finally, two classes of communication faults should be considered - (1) loss of connectivity (local area network or wide area network); and, (2) overloaded or congested communications.

The SDCE should also provide some support for real-time applications. It will enable distributed applications to incorporate scheduling and resource decisions based on the current conditions of both system and environment within some specified time constraint.

A base for adaptive resource management should be supported to improve throughput and provide some level of fault avoidance. This base should support both static and dynamic migration of processes and files.

Support for multi-domained applications is also desirable. This would permit controlled access to resources (e.g., computers and data) in a flexible and efficient manner.

Support for these requirements through services such as processes, IPC, resource management, file servers, etc., has typically been provided by network and distributed operating systems. More recently, however, micro-kernel technology provides a new, more modular, and adaptable approach towards meeting these requirements.

3.0 Micro-kernels

Traditionally, network and distributed operating systems have been implemented by spreading knowledge about the system throughout large monolithic kernels. This monolithic design complicates the task of developing and integrating advances in technology with respect to both hardware and software. Fortunately, recent trends in operating systems design have led to the use of micro-kernel architectures. This approach separates the components of the operating system that control hardware resources from those that determine the flavor of the operating system environment, e.g., a given file system interface. This allows the most complex software layers to be built on top of a relatively simple kernel (Figure 1).

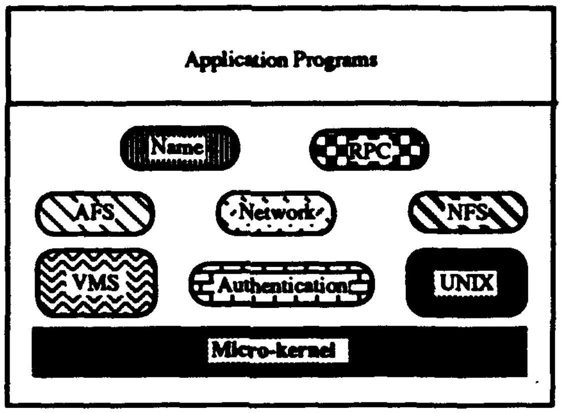


Figure 1. Micro-kernel Architecture.

Due to this modular layered approach to operating system design, a micro-kernel architecture is able to offer a number of advantages over traditional network and distributed operating systems. Some of these advantages include : *tailorability* - the operating system environments can be customized for specific applications; *portability* - the operating system environment code is independent of a machine's instruction set, architecture, and configuration; *extensibility* - additional operating system environments and versions can be incorporated alongside existing systems; *real-time* - support for real-time applications is possible since the kernel is no longer required to hold long interrupt locks for UNIX system services; *multi-processor support* - since the kernel is not required to support complex system functions (which may limit parallelism), greater parallelism is possible for its functions; furthermore, the micro-kernel's features can be better tailored to parallel applications; *multi-*

computer support - since the kernel only provides a relatively small number of basic abstractions, it can optimize the mapping of each abstraction onto the distributed hardware; and *security* - a smaller kernel is more easily defined and implemented in a secure manner; this modular, layered architecture is simply better suited to trusted systems than that of traditional monolithic kernels [Black].

While the current level of maturity for micro-kernel technology does vary, some systems (e.g., Mach and Chorus) are already achieving commercially competitive levels of functionality and performance.

4.0 Mach and Chorus

Two candidate micro-kernel architectures were evaluated in this study - Mach [Black] and Chorus [Armand].

Some of the key features which each of these architectures have in common include: (1) a small kernel; (2) a modular architecture which provides scalability and allows dynamic configuration of the system and its applications; (3) transparent network access for interprocess communication; (4) a communication-based architecture which implements generic services used by a set of subsystem servers to extend standard operating system interfaces; (5) support for concurrency in both the operating system services and application programs; (6) support for large address spaces with flexible memory sharing; (7) integration of message passing communication with virtual memory; (8) real-time support which is accessible by system programmers; and, (9) UNIX support.

4.1 Chorus

Chorus was a distributed systems research project that was conducted in France from 1979 - 1986. Three versions were developed. They are known as Chorus-V0 (1980-1982), Chorus-V1 (1982-1984), and Chorus-V2 (1984-1986). Chorus-V3 (1987-present) is the current version (and the one we refer to in this paper) and it is designed to integrate the best features of all the previous versions of Chorus. Chorus was designed with the intention of supporting industrial quality operating system environments.

The main abstractions implemented by Chorus include *actors*, *threads*, and *ports*. An actor is a collection of resources in a Chorus system. An actor defines a protected address space supporting the execution of threads that share the resources of the actor. A thread is the unit of execution in the Chorus system. A thread is a sequential flow of control and it is always tied to exactly one actor which defines the thread's execution environment. Within an actor, multiple threads can be created and can run concurrently. A port represents both an address to which messages can be sent and an ordered collection of unconsumed messages. When created, a port is attached to a specified actor. Only threads of this actor may receive messages on that port.

4.2 Mach

The Mach 3.0 micro-kernel architecture was developed at Carnegie Mellon University. The history which led up to its development includes RIG (1976-1981), which led to Accent (1981-1986), which in turn was followed by the Mach 2.5 operating system (1986 - 1989). The Mach 3.0 micro-kernel (1989 - present) evolved from the Mach 2.5 operating system.

The basic abstractions of Mach are the *task*, *thread*, and *port*. A task may be viewed as a container to hold references to resources in the form of a port name space, a virtual address space, and a set of threads. A thread is an execution point of control. It is the basic computational entity. It belongs to one and only one task that defines its virtual address space. A port is a unidirectional communication channel between a client who requests a service and a server who provides the service.

4.3 Chorus IPC VS Mach IPC

Mach and Chorus abstractions are very similar with respect to resource management (Chorus' actor and Mach's task), control (threads), and virtual memory. The addressing and communication abstractions of Mach and Chorus are, however, quite distinct.

In both Chorus and Mach, messages are addressed to intermediate entities called *ports*, not directly to threads or actors/tasks. It is the port abstraction that provides the necessary decoupling of the interface of a service and its implementation. This provides a basis for dynamic reconfiguration. Consequently, a port can migrate from one actor/task to another.

Addressing in Chorus is accomplished in a global manner via unique identifiers (UI). All Chorus objects (e.g., actors, ports) are referenced in this manner. The Chorus micro-kernel implements a UI location service which allows the referencing of Chorus objects without knowledge of their current location. In contrast, Mach does not support the notion of global addressing, i.e., all port's rights resolve to local ports. A network server extends Mach IPC across the network via the use of local *proxy* ports to represent remote ports. This collection of network servers (one on each node) then maintains the current location of network-wide ports.

The Chorus inter-process communication (IPC) mechanism permits threads to communicate via unreliable asynchronous point to *port group* (a port group is an abstraction which extends message passing semantics between threads by allowing messages to be directed to a group of threads), or by synchronous reliable remote procedure call (RPC). When messages are sent to port groups, it is possible to: (1) broadcast to all ports in the group; (2) send to any one port in the group; (3) send to one port in the group located at a given site; and, (4) send to one port in the group located on the same site as a given UI. Note, however, that receive semantics are limited to one-to-one. That is, a port can only receive messages from a single sender (Figure 2a).

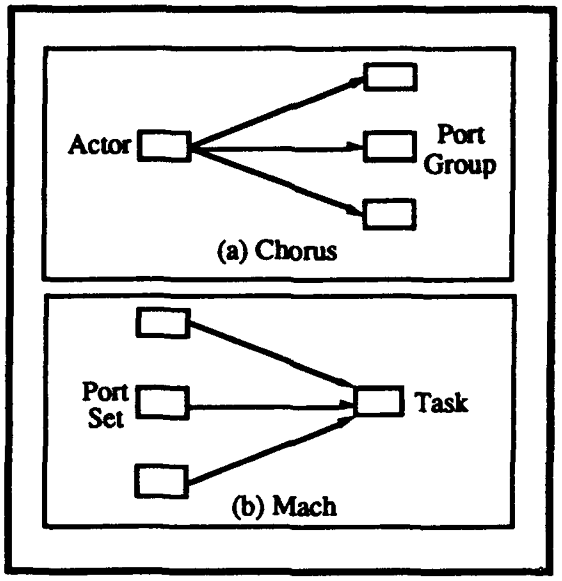


Figure 2. Send/Receive Semantics.

In contrast, Mach provides reliable point to point communication. With respect to send semantics, the notion of a multicast is not present. The receive semantics, on the other hand, permit a task to receive messages from (potentially) multiple senders, known as a port set (Figure 2b).

5.0 Micro-kernel Selection

The SDCE technology base (with its supporting technologies) must be capable of meeting all the SDCE requirements. This base must also be capable of evolving with the needs and advancing technologies of C2 systems. Furthermore, since the micro-kernel technology is the foundation for the SDCE, it must be a relatively mature and stable system.

Overall, the mechanisms provided by Chorus and Mach are comparable. The main difference lies in the IPC mechanism, as mentioned previously. Each of these IPC mechanisms have their strengths and weaknesses. Clearly, an ideal solution would be to combine the strengths of each of these abstractions. This very concept is nearing completion at Cornell University [Glade]. Work has also been done to improve the speed of Mach's IPC mechanisms [Draves] [Barrera III]. In addition, Real-Time Mach and Distributed Trusted Mach are maturing along with the development of the Mach micro-kernel. Furthermore, it is clear that Mach is becoming an industry standard (e.g., Open Software Foundation). Due to these considerations, the Mach micro-kernel has been selected for the SDCE foundation.

Of course, the Mach micro-kernel by itself does not meet all of the requirements of the SDCE. Therefore, we now present several different architecture design options that include supporting technologies (built on Mach or the Mach micro-kernel) which can meet the requirements of the SDCE. Isis and Cronus are two such supporting technologies. They provide support for distribution, fault-tolerant mechanisms (replication, etc.), limited adaptive resource management, multi-clustered networks, reliable communication, virtual synchrony, and concurrency.

6.0 SDCE Architecture Options

The first architecture to be considered is built on the Mach 2.5 operating system (Figure 3). This architecture is viewed as the easiest and lowest risk option because it is based on proven technology. However, it is not a micro-kernel based design and, therefore, it would be more difficult to adapt this system to the long term, ever-changing needs of C2 systems.

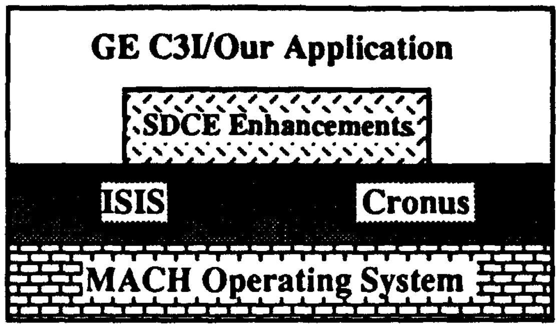


Figure 3. SDCE Architecture 1.

Unlike the architecture in Figure 3, the second architecture design option (Figure 4) utilizes the Mach micro-kernel. This architecture, however, requires an intermediate UNIX server in order to integrate the functionality provided by Isis and Cronus. Although this architecture is micro-kernel based, it is not completely faithful to the micro-kernel design philosophy as illustrated in Figure 1. That is to say, Isis and Cronus interact with the Mach micro-kernel through the UNIX server rather than directly with the Mach micro-kernel itself. This level of indirection is due to the fact that Cronus and Isis were originally developed in a UNIX environment.

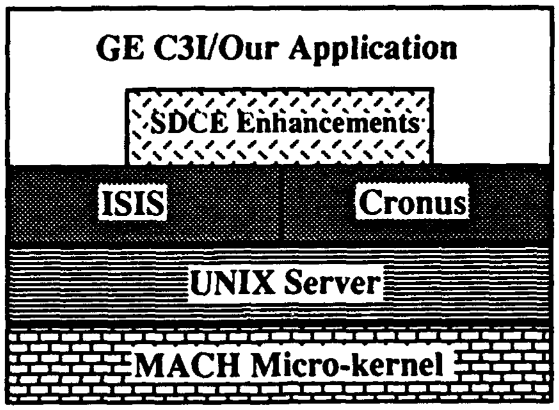


Figure 4. SDCE Architecture 2.

The third option, (Figure 5), truly implements the micro-kernel design philosophy. Horus (son of Isis) consists of an Isis Toolkit [Birman] and IPC enhancements [Glade] to the Mach micro-kernel. Consequently, the need for an intermediate UNIX server is no longer required as an interface to the Mach micro-kernel. However, the UNIX server is still utilized by Horus as a development environment by providing services such as file systems, editors, compilers, debuggers, and so forth. This work is nearing completion at Cornell University.

In addition, work is also being planned to build Cronus directly on top of the Mach micro-kernel. When available, this capability could easily be incorporated into this architecture. In the interim, however, Cronus could rest on top of the UNIX server as shown in Figure 5.

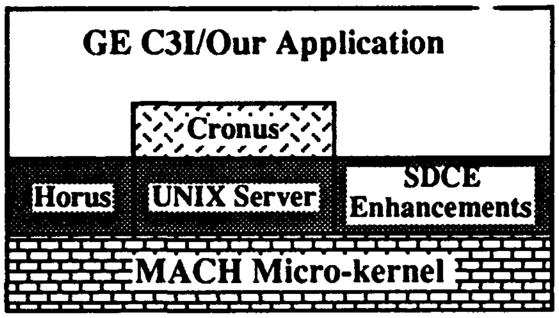


Figure 5. SDCE Architecture 3.

Figure 6 presents a hardware solution to the fault-tolerant requirements of the Survivable Distributed Computing Environment. The Fault Tolerant Multi-processor (FTM) micro-kernel extends Mach by adding mechanisms which allow implementation of fault-tolerance. Normal Mach entities can be corrupted by a processor failure. The FTM architecture can tolerate any single hardware fault.

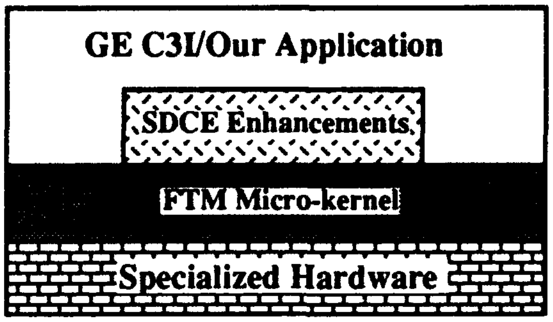


Figure 6. SDCE Architecture 4.

The SDCE enhancements in Figures 3 - 6 may include a reliable transaction service (e.g., Camelot), a replicated file server, and an X.500 server [Weider]. The GE C3I application that is noted in Figures 3 - 6 refers to an Adaptive Fault Resistance System (AFRS) effort by General Electric that is currently under development to provide higher degrees of availability, survivability, and graceful degradation than is currently available in non-adaptive systems.

As previously mentioned, the architecture represented in Figure 5 is more true to the micro-kernel design, and provides the most versatile base on which to build survivable distributed applications. It is, therefore, the architecture of choice for supporting the SDCE requirements. However, it should be noted that the risk factor at this time may be relatively high for this architecture option due to the immaturity of Horus. In the event that Horus is too immature to be used within the allotted time frame, the architecture represented in Figure 4 will be utilized instead. This does not preclude the incorporation of Horus at some future time.

8.0 Acknowledgments

The authors of this paper would like to acknowledge Dr. Gary L. Craig (Syracuse University) for his useful discussions in micro-kernel technology. We would also like to thank him for his comments in reviewing this paper.

9.0 References

Armand, F., et. al., Towards a Distributed UNIX System - The Chorus Approach, *Proceedings of the European UNIX Systems User Group Conference*, September 1986.

Barrera, Joseph S. III, A Fast Mach Network IPC Implementation, *Usenix Association*,

Birman, Kenneth P., The Process Group Approach to Reliable Distributed Computing, Department of Computer Science, Cornell University, July 1991.

Black, David L., et. al., Micro-kernel Operating System Architecture and Mach, *Usenix Association*, pp. 11-13.

Cooper, Robert C. B., Glade, Bradford, B., Birman, Kenneth P., and Robbert van Renesse, Light-Weight Process Groups, Department of Computer Science, Cornell University, 1992.

Draves, Richard, A Revised IPC Interface, *Usenix Association*.

Glade, Bradford B., Birman, Kenneth P., and Robbert van Renesse, Group Communication in Mach: Kernel Interface Supplement, Department of Computer Science, Cornell University, November 3, 1992.

Weider, Teynolds, and Heker, Technical Overview of Directory Services Using the X.500 Protocol, NIC RFC 1309, 16 pages, March 1992.

Weak Consistency and Recovery in a Command and Control Environment

Anthony M. Newton
Computer Systems Branch (RL/C3AB)
Rome Laboratory
Griffiss AFB, NY 13441-5700

ABSTRACT

In a distributed system, access to resources is considered to be one of the most important attributes. Availability is most often obtained by replicating resources throughout the distributed system. Once replicated, resources must be maintained consistently. Consistency is a measure which attempts to describe the variance in data when accessed from two (different) replicated sites. Strong consistency is characterized by zero variance access between copies. Weak consistency is characterized by the possibility of retrieving a copy which (typically) has not been updated. Inconsistency manifests any time that copies do not contain the same data. Recovery attempts to bring inconsistent copies back to a consistent state. Most command and control (C2) applications need the properties of a distributed system because, in a wartime environment, application survival in the presence of nodal failures and network partitions is most desirable. The problem with strong consistency in this environment is that the exercise required to enforce strong consistency may either: (1) consume more processor cycle resources than are appropriate for the data or (2) result in delays obtaining the data sufficient to invalidate its use. This paper summarizes a proposed method for building weak consistency policies from recovery mechanisms in an object-oriented system suitable for the C2 environment. The full paper was presented at the NATO Workshop of Research symposium on Object Oriented Distributed Systems held 12-15 May 1992. The full paper may be obtained from the author.

INTRODUCTION

In a distributed system, available access to stored resources (e.g. data) is considered to be one of the most important attributes. Availability is most often obtained by replicating data throughout the distributed system. However, replicating (copying) the data is not sufficient. As data is replicated, maintaining consistency becomes a dominant concern.

Consistency is a measure which attempts to describe the variance in data content when accessed from two (different) replicated sites. In a single copy case, the data accessed is presumed to have no variance from the stored data. When multiple copies exist, a change to one copy makes the other copies inconsistent. The change must be propagated to the other copies before access is granted to all copies if a uniform view of the data is desired. The level of consistency is defined by the type of access granted to the data copies when state changes are en route.

Strong consistency is characterized by zero variance access between copies. In general, systems which employ strong consistency mechanisms will guarantee that access to copies is denied until all copies of the data have been updated of any change. This may result in data access times that either (1) consume more processor cycle resources than are appropriate for the data or (2) result in delays sufficient to invalidate its use.

To help relieve this problem, some of the requirements necessary for strong consistency were relaxed (weak consistency). Typically, weak consistency is characterized by the possibility of retrieving a copy which has not been updated, but refusing a write or update request to a copy until any current changes have been distributed to all copies. This gives greater read availability, yet still restricts the write availability.

Most C2 applications need the properties of a distributed system because, in a wartime environment, application survival in the presence of nodal failures and network partitions is most desirable. Enforcing a policy of strong consistency within a distributed command and control application can have disastrous effects, because of the time necessary to guarantee data consistency in either read or write modes, or because of deadlocks which may occur and immobilize processing. Traditional weak consistency mechanisms may improve the read/write performance time and may allow for operation on one side of a network partition, but, C2 applications are likely to need operational capability on both sides of a partitioned environment. In order to allow such operation, it will be necessary to reexamine issues of serialization and inconsistency.

APPROACH

If multiple copies are presumed, inconsistency manifests any time that the copies do not contain the same data. Thus, a change to one copy results in an inconsistent state which is made consistent by propagating the changes to all other copies. So, recovery can be defined as the attempt to bring inconsistent copies back to a consistent state.

The intent of this paper is to describe a system that is being developed to facilitate experimenting with recovery mechanisms that allow both read and write operations on copies of data that are separated by a network partition. The base of the system will be comprised of an object-oriented distributed system. Object management mechanisms will be added or altered to support partitioned operation, recovery mechanisms, and an application interface to be used in the exchange of information that should occur during a recovery operation.
The system being proposed rests on the following set of assertions.

Partitioned Data Exists without Networks

Organizations partition data and resources to allow for decentralized control and operation. Thus, most activity occurs within a local area, with read/status/update request coming from outside that local span of control. When network partitions occur, the data involved will tend to reflect the conditions present on the respective sides of the partition as local applications continue to process information.

Old Data and Incorrect Data are Distinct

Contrary to views that apply to non-partitioned, quick-access environments, there is a distinction between old data and incorrect data. Incorrect data will be defined as data that has been corrupted in some manner. In general, this corruption is detectable and techniques exist for its correction. Old data will be defined as correct data that lies within a time line, since all current data becomes old data over time. The time difference does not make the data incorrect but merely out-of-date.

Applications Tolerate Old Data

All distributed applications tolerate old data to some degree, since networked systems cannot deliver instantaneous access to data. Moreover, any current data is likely to change after an application completes a successful read operation.

Communication Failure

When communication is lost with a copy, two events could have occurred. The computer node containing the copy could have failed, or, a network partition could be isolating the computer node. A network partition is defined as a loss of communication between copies that need to interact, however, a network partition is recognized only when two copies have states that disagree for a specific version. Therefore, updates are the trigger mechanism for determining network partitions and initiating recovery procedures.

OBJECT MODEL

The object model is an abstraction used by both programmers and system designers to facilitate a view of the data within a computer system [Jones79]. In its most basic description, an object is a unit of data. As such, the object is entirely passive in nature. Operations are code segments that interact with the object to either change or report the state of an object. The binding of an operation to an object is called an invocation.

The active components of object-oriented systems can be represented in several ways. In some cases, active entities called managers and clients interact with the objects to form application tasks [BMS85], [Vinter89], [Honey88]. In other object-oriented systems, the active entities are called threads, where threads move through objects and constitute a single line of processing [North88], [GIT89], [Pitts88], [Dasgu88], [Tevan87].
The system described in this paper uses the object-manager representation to discuss system procedures and data. Objects are passive and are contained within the manager address space. The manager is comprised of tasks, which only operate within confines of the manager's address space. These tasks can either be long-lived, such as daemons, or be limited in duration to the operation that was assigned to the task. Tasks relate to objects by copying the object data from persistent storage, operating on the data, then returning the modified version to persistent storage. The procedures that will be defined to handle access, update notification, and recovery of replicated objects exist inside the persistent storage interface.

While not stated, it should be implied throughout this paper that appropriate synchronization (locking) of common resources is provided in cases where true concurrent task execution is permitted (e.g. multiprocessors). Applications are built from object managers.

CONSISTENCY MANAGEMENT

Primary/Secondary Object Technique

The physical method by which objects are distributed throughout the system uses the idea of a primary copy. This asserts that one object is designated as the PrimaryCopyObject. All other instances of the object will be known as a SecondaryCopyObject. Any processing required for operations on the PrimaryCopyObject must maintain the correct view of an object and update the other secondary copies. This is enforced through the requirement that all read and write operations must be performed on the PrimaryCopyObject. Increased availability could be gained by designating a new PrimaryCopyObject if the original was inaccessible and a majority of the remaining copies were available. Weakened consistency rules allowed read operations to occur on either the primary or secondary object [David85].

For the method proposed within this paper, not only is SecondaryCopyObject reading and PrimaryCopyObject election used, but an extension has also been added which allows a PrimaryCopyObject to become established on each side of a partition. Having multiple PrimaryCopyObject sites gives the system the flexibility of operation within partitioned environments but places a greater burden on the recovery procedures.

Updates
An update log is maintained on the PrimaryCopyObject which details any changes made to the object. The log is written each time a write operation is performed on the PrimaryCopyObject, or a remove operation for the object is processed. A flag in the log entry determines whether the entry is for an update or a removal. The log entry also contains the new or updated contents of the object data, the identifier for the object, the locations of SecondaryCopyObjects which have not been updated, and the originator of the operation request. Periodically, update operations are issued to SecondaryCopyObjects, based on the current version of the PrimaryCopyObject. Update operations move SecondaryCopyObjects into a recovery state, where procedures are initiated to bring the SecondaryCopyObject consistent with the PrimaryCopyObject. Entries in the update log are normally removed once all copies have been updated or removed. Any write operation that successfully completes will notify the originator of the operation.

Copy Control
Another aspect of the consistency management process is the maintenance of the number of object copies that exist in the system. When an object is created, the creator determines the maximum number of copies that should be maintained. If the total number of objects, including the PrimaryCopyObject, is smaller than this maximum, the consistency process attempts to create SecondaryCopyObjects to satisfy the requirement. This process could also be linked with resource management processes in order to satisfy higher order requirements of availability and performance.

Failure
When a partition is recognized during a write operation, a new PrimaryCopyObject must be designated on the side(s) of the partition without one. The new PrimaryCopyObject and any other SecondaryCopyObjects operate independently within the domain created by the partition.

When either part or all of the partition is resolved, recovery procedures are activated to merge the copies into a consistent object resulting in one PrimaryCopyObject. A partition is resolved if communications are reestablished between object copies. A SecondaryCopyObject receiving an update notification listing a PrimaryCopyObject other than the established one, knows that it has moved from one side of a partition to another. The SecondaryCopyObject moves into a recovery state, where it attempts to become consistent with environment of the new partition. Any PrimaryCopyObject which receives an update notification, knows that another PrimaryCopyObject exists within the partition. An election procedure is initiated to resolve the conflict and to assign the object responsible for carrying out the recovery process. During the recovery process, the version vectors that are part of each object determine whether copies have become inconsistent. The update logs from each PrimaryCopyObject serve as a tool which can be used during the recovery process.

Recovery
One of the views being presented in this paper asserts that consistency is a high-level goal achieved through recovery procedures. A change to any one of the object copies results in inconsistency. A proper way to recover from the inconsistency is to spread the updates to the other copies, making them consistent. Strong consistency is a policy that enforces a refusal to read any object copy until the recovery mechanism (in this case an update procedure) has made all of the copies consistent.

a nominee is available to be conquered

Level	Mechanism	Description
Object	StoreObjectState	Write object state to non-volatile storage
Object	RejectObjectState	Flushes last object state from memory
Object	ObjectStateReplace	Replace memory object state
Object	ObjectStateCheck	Test operation against conditional state
Object	ObjectStateMerge	Combine two object states into one
Object/System	AuthorityObject	All operations must pass through this object
System	SerialByCopy	Apply each set of operations based on object
System	SpreadByCopy	Apply each set of operations evenly on objects
System	CopyElection	Replace election algorithm
System	VectorDominance	Test version vectors

Table 1: Recovery Mechanisms

The system being described is attempting to enforce an object level separation between the policies that govern a system (e.g. consistency, availability, atomicity, etc.) and the recovery mechanisms. This will allow both system and application procedures to tailor the needs of consistency without requiring that all objects in the system adhere to any one policy. During the first level of experimentation, the system will provide a set of recovery mechanisms used to implement some consistency issues (see Table 1).

ELECTIONS

Elections are required whenever the system needs to designate a PrimaryCopyObject. While a static algorithm could be chosen to dictate PrimaryCopyObject choices, this solution would not be very practical or flexible, since all location sites would have to be known in advance (either at compile time or at the start of run time). The method of selecting a new PrimaryCopy, discussed below, is primarily based upon the robust election protocol of [Kim88].

The election process begins by nominating an object to become the PrimaryCopyObject. Once an object has been nominated, the election process begins to acquire a set of other objects which are conquered by locking them. Any SecondaryCopyObject which is not and locked. A PrimaryCopyObject is an automatic candidate. Locked objects are added to the nominee's list of conquests. If the election process attempts to lock another candidate, the winner of the contest is the nominee (in order of importance): with a special application designator, who already was the PrimaryCopyObject, with the most locked copies, or with the luck of a coin toss. Conquered nominees give up their acquired objects to the conqueror and are then locked by the conqueror. The ultimate winner is determined either by locking all copies or by time-out value.

The modified robust protocol used in the system provides for reliable operation in the presence of network partitions, but it does have two weak areas. The protocol may require several iterations before electing a PrimaryCopyObject, since the protocol does not prevent the formation of multiple PrimaryCopyObjects if network partitions appear during an election and disappear afterward. The management process detects multiple PrimaryCopyObjects and results in another election being held to resolve the problem. The protocol can become deadlocked if network partitions continue to appear and disappear during either the election or recovery procedures. This is due to the reliance on time-out values. This weakness is not considered debilitating because network failures of this type are considered unlikely.

This presumption can be defended if the environment is constructed using integrated sets of computer resources locally connected to each other and remotely connected through some wide-area communications transport. In a combat situation, it is likely that partitions will occur. It is also likely that those partitions are either planned (with stable, predictable connect times) or unplanned (probably the result of some set of resources being destroyed). Planned partitions can ensure that the connect times are sufficient to properly reintegrate the nodes. Unplanned, catastrophic partitions will not be worth the effort.

OBJECT STATES

Read/Write Access

All access to objects can reduce to two simple operation classes: read and write. However, because there are multiple copies of an object, and because the actions that result from a consistency update procedure are different than a simple write procedure, three operation classes are used: read, write, and update.

In general, all read operations will complete successfully if the object exists. The read operation provides an optional parameter indicating the quality preference of the read. If "high quality" is preferred, the read will be carried out on the PrimaryCopyObject, otherwise, a SecondaryCopyObject will be used to perform the operation. High quality reads which cannot be satisfied will downgrade to reading a SecondaryCopyObject. The data returned from the read is always tagged SubjectToRecovery due to the possibility of an update occurring during transit time. The data received from a read operation will not be validated by the system as OK at some later time.

Write operations are always performed on the PrimaryCopyObject. An election is held in cases where either failure or network partition is recognized to designate a SecondaryCopyObject as a PrimaryCopyObject. Write operations are always considered to be SubjectToRecovery unless only one copy of the object exists. Changes to the object are made to a memory copy until explicitly requested to be written to non-volatile memory. Writes are made permanent, rejected, or refused based upon the decision of the recovery procedure. Anything receiving a SubjectToRecovery status code for a write operation will, ultimately, be informed of the final status of their write after recovery procedures are complete.

FieldName	ValueType	Description
PrimaryCopy	Boolean	Am I PrimaryCopyObject
AuthoritarianCopy	Boolean	Am I an Authoritative Source
ProlifCount	Integer	Ideal # of copies within system
UpdateLog	List of log entries	Current update level
RemoveCopies	Boolean	Remove object from system
PCLocation	Address	Location of PrimaryCopy
CopysAvailable	List of location	Locations of communicating objects
CopyLocations	List of (location/update)	VersionVectors

Table 2: Object Management Block (OMB)

Update operations are only performed on SecondaryCopyObjects. Consistency procedures ensure that changes in PrimaryCopyObject state are applied to SecondaryCopyObjects through the update operation. A network partition is suspected whenever updates cannot be delivered to SecondaryCopyObjects. A network partition is verified whenever an update operation is applied to the

PrimaryCopyObject. The only way to resolve the state of the PrimaryCopyObject is to hold an election among the copies so that recovery mechanisms can start.

Current State	Transition Event	Next State
<null>	Object Start	Recovery
Normal	Notification of Inconsistency	Recovery
Normal	Election Candidate	Genghis
Normal	Election Participant	Locked
Genghis	IAmSuperior Winner	<same state>
Genghis	IAmSuperior Loser	Locked
Genghis	Internal State Time-out	Recovery
Locked	Internal State Time-out	Genghis
Locked	Election Complete	Recovery
Recovery	Recovery Complete	Normal
Recovery	Object Exit or Destruction	<null>

Table 3: Object Operation State Table

Object Management Information
To assist in maintaining consistency, a record is associated with each object that contains replication management information (see Table 2).

All objects run in one of several states: Normal, Genghis, Locked, and Recovery (see Table 3). Each name implies the operational condition of the object.
The NORMAL state is used by the object for general operation. The Normal state can only be entered from the Recovery state. Movement from the Normal state can occur as either the result of an election process or through an update directive from the consistency management process. Elections transition the object into the Genghis or Locked state. Update requests move the object into a Recovery state. The Normal state is the only state where read and write access to the object is granted.

The RECOVERY state is used to make a SecondaryCopyObject consistent with the appropriate PrimaryCopyObject. It is entered through any one of three conditions: an initial condition resulting from object start/creation, from the Normal state as a result of an update condition, or from either Locked or Genghis states after an election. While in the Recovery state, the object is manipulated according to the conditions encountered when the object entered this state. The object transitions out of this state by the completion of the recovery procedures. The only valid state transition from the Recovery state are either to the Normal state or to shutdown (terminate or destroy).

The GENGHIS state is entered when an object operation has detected the need to elect a new PrimaryCopyObject. This applies under the following conditions: (1) the existence of two PrimaryCopyObjects resulting from a partition, (2) a SecondaryCopyObject performing a write operation because the PrimaryCopyObject is unreachable, or (3) a timeout expiration due to an object left in the Locked state as a result of a partition during the election. To the object, being in the Genghis state represents a nomination to PrimaryCopyObject status. While within the Genghis state, the election process adds object locations to the list of available and locked objects. Movement out of the Genghis state occurs either when the election process has determined that another object is the best nominee or when the election process reaches a time-out value declaring the current Genghis candidate to be the new PrimaryCopyObject. If the election

process chooses another candidate, the object is moved from the Genghis state to the Locked state. If the object in the Genghis state is chosen to be the new PrimaryCopyObject, it is moved to the Recovery state.

The LOCKED state is used only in the election process. Objects which are placed in the Locked state during an election process remain there until either the election is over or a time-out is reached. If the new PrimaryCopyObject is not chosen within a predetermined amount of time, a daemon process will start an election process and move any objects in the Locked state to the Genghis state, where they become candidates for PrimaryCopyObject. This safeguards against network partitions causing lock-out during the election process. Objects in the Locked state will be moved into the Recovery state if the election was successful.

FINAL COMMENTS

This paper is intended to provide a differing perspective and to provoke debate on the worthiness of providing absolute consistency in a military system where nodal failures and partitioning of resources are real events. The design presented herein provides a framework for experimentation in this area.

REFERENCES

[Jones79] Jones, Anita K., et al., "The Object Model: A Conceptual Tool for Structuring Software", OPERATING SYSTEMS: An Advanced Course, Springer-Verlag, New York, 1979, pp 8-16.

[BMS85] Berets, James C., Mucci, Ronald A., and Schantz, Richard E., "Cronus: A Testbed For Developing Distributed Systems", IEEE Military Communications Conierence, IEEE Communications Society, October 20 - 23, 1985.

[Vinter89] Vinter, Stephen T., "Integrated Distributed Computing Using Heterogeneous Systems", SIGNAL, June 1989.

[HONEY88] Badarinath, N, et al., "Fault Tolerant Distributed Systems: Final Technical Report", RADC-TR-88-158, Rome Air Development Center, August 1988.

[GIT89] LeBlanc, R., et al., "Action Based Programming for Embedded Systems", RADC-TR-88-295, Rome Air Development Center, February 1989.

[PD88] Pitts, D., Dasgupta, P., "Object Memory and Storage Management in the Clouds Kernel", Proceedings: Eighth International Conference on Distributed Computing Systems, June 1988, pp 10-17.

[Pitts88] Pitts, D., "Recovery in the Clouds Kernel", Proceedings: Seventh Symposium on Reliable Distributed Systems, October 1988, pp 167-176.

[Dasgu88] Dasgupta, P., et al., "The Clouds Distributed Operating System", Proceedings: Eighth International Conference on Distributed Computing Systems, June 1988, pp 2-9.

[Tevan87] Tevanian, A., et al., "Mach Threads and the UNIX Kernel: The Battle for Control", Proceedings: 1987 USENIX Conference, June 1987, pp 185-197.

[North88] Northcutt, J.D., et al., "Decentralized Computing Technology for Fault-Tolerant, Survivable C3I System: Functional Description", PIIN F30602-85-C-0274, CDRL A004, Carnegie Mellon University, December 1988.

[David85] Davidson, S., et al., "Consistency in Partitioned Networks", ACM Computing Surveys: Volume 17, Number 3, September 1985, pp 341-370.